Implementation of an automated eye-in-hand scanning system using Best-Path planning

Jordi Rebull Mestres

Advisor: Mattia Racca

Professor: Ville Kyrki

September 22, 2017

Master's Thesis

Double MSc in Robotics and Industrial Engineering

Acknowledgements

This thesis has been carried out in the Intelligent Robotics research group at the Aalto University, Finland, during an ERASMUS+ exchange program. In the five months spent in Helsinki, I had the possibility to work in an exciting and new environment, surrounded by kind international people always willing to help. Among the member of the research group I would like to thank my supervisor prof. Ville Kyrki and advisor Mattia Racca for their guidance and support. Ville gave me the opportunity to work on this interesting implementation project that allowed me to learn a lot about Linux, ROS and C++. Mattia was always available to help, and thanks to him the project kept going on when implementation errors blocked the way. Finally, I want to thank my labmates for the indispensable teamwork, specially to Edoardo, with whom I spent a great time and supported me with our large discussions about our respective thesis.

Espoo, 18.07.2017

Jordi Rebull Mestres

Abbreviations and Acronyms

ROS	Robotic Operative System
KUKA	Keller Und Knappich Augsburg
LWR	Low Weight Robot
MATLAB	MATrix LABoratory
NBV	Next-Best-View
NBS	Next-Best-Scan
BP	Best-Path
DOF	Degrees Of Freedom
GUI	Graphical User Interface
ToF	Time-of-Flight
ICP	Iterative Closest Point
IR	InfraRed
RGB-D	Image channels: Red, Gren, Blue - Depth
TSDF	Truncated Signed Distance Function
GPU	Graphics Processing Unit
CPU	Central Processing Unit
RAM	Random-Access Memory
CUDA	Compute Unified Device Architecture (Nvidia)
RRT	Rapidly-Exploring Random Tree
RRTConnect	Bi-directional RRT
POI	Point Of Interest
YCB	Yale-CMU-Berkeley
TF	ROS Transformation Frame
FRI	Fast Research Interface (library)

Contents

Abstract

In this thesis we implemented an automated scanning system for 3D object reconstruction. This system is composed of a KUKA LWR 4+ arm with Microsoft Kinect cameras placed on its extreme and thus, in an eye-in-hand configuration.

We implemented the system in ROS using *Kinect Fusion* software with extra features added by R. Monica's previous work [16] and *MoveIt!* ROS libraries [29] to control the robot movement with motion planning. To connect these nodes, we have coded a suite using ROS and MATLAB to easily operate them as well as including new features, such as an original view planner that outperforms the commonly used Next-Best-View planner. This suite incorporates a Graphical User Interface that allows new users to easily perform the reconstruction tasks.

The new view planner developed in this work, called Best-Path planner, offers a new approach using a modified Dijkstra algorithm. Among its benefits, Best-Path planner offers an optimized way to scan the objects preventing the camera to cross again the areas which have already been scanned. Moreover, viewpoint location and orientation have been studied in depth in order to obtain the most natural movements and get the best results. For this reason, this new planner makes the scanning procedure more robust as it assures trajectories through these optimized viewpoints, so the camera is always looking towards the object maintaining the optimal sensing distances.

As this project is focused on its later utility in the Intelligent Robotics Laboratory, we uploaded all the source code in the Aalto GitLab repositories [37] with installation instructions and user guides to show the different features that the suite offers.

1 Introduction

1.1 Motivation

Autonomous 3D modelling has been an important field of research in the last years. Acquiring 3D models of real objects is essential in a lot of different industrial applications, such as rapid prototyping or reverse engineering. Moreover, in robotics field it is very useful for applications such as object recognition, grasping and manipulation or collision detection.

Some years ago all the range cameras available for 3D object modelling were specialized in certain applications and were very expensive. However, when low-cost RGB-D sensors such as Microsoft Kinect appeared in the market everything changed: 3D modelling became available for everyone and it sparkled a renewed interest in software tools to digitize objects.

The motivation of this project is to better understand how 3D reconstruction works and implement a complete system to test, evaluate and improve it. The system will use a low cost *Kinect* sensor depth camera and an open-source software, *Kinect Fusion* [18], for the reconstruction part so this platform can be useful for future projects as it will be very accessible. The system will be mounted in an eye-in-hand configuration attaching the camera to the tool centre point of a robotic arm.

The aim of this project is to make a robust and accessible autonomous 3D object scanning configuration for grasping purposes. Although the original idea of this work was to implement the reconstruction system on the arm of a mobile robot, it is being implemented on a KUKA robot arm as the initial development. Nevertheless, the complete system can be later adapted to different platforms so this first approach is equally valid.

1.2 Research Objectives

The main research objectives that are proposed to achieve in this thesis are the following:

- Implement the *Kinect Fusion* software [18] [16] and build a Next-Best-View algorithm (NBV).
- Develop a new view planning algorithm based on trajectory planning, outperforming the original NBV.
- Build a software and hardware environment to run the reconstruction programs and the planning algorithms, to test, analyse and compare them.

1.3 Thesis Structure

This thesis is organized as follows: In next Section 2, related work in the field of automated 3D reconstruction is discussed as well as the current technologies used in this project. By this way the background, the current methods to perform 3D reconstruction and our new methods are presented in this section.

Then, in Section 3 the system implementation is explained. In this section the complete software structure is shown to understand how the different components of the code work together. The ROS nodes implementation of the used programs (*Kinfu* and *MoveIt!*) are commented, as well as all the original ROS and MATLAB nodes that connect them and perform the view planning algorithms.

After the system implementation, the experimental results are explained in Section 4. The used set-up for all the tests is shown, as well as all the different configurations for all the experiments. Then, the results of both view planning algorithms are shown and compared, opening a discussion on their highlights and drawbacks.

Finally, last Section 5 of conclusions contains the final thoughts about the project, the reached objectives and the future work in this field.

2 Automated 3D Reconstruction

In this section, the state of the art of automated 3D reconstruction is given as well as the technology and methodology used in this project. By this way, the reader is able to understand the current work performed in this topic and where the project stands with respect to the general development.

2.1 Related Work

Active vision perception, concretely object reconstruction, is the acquisition of a virtual computer model of the surface of a physical model. The virtual model is usually composed of a triangulated polygonal mesh that represents the surface geometry (Figure 1). If this object reconstruction is wanted to be automatized, several components are needed to be set in the system, such as a positioning system for the sensors or the objects and a planner to look for the next best sensor position that moves there avoiding collisions.

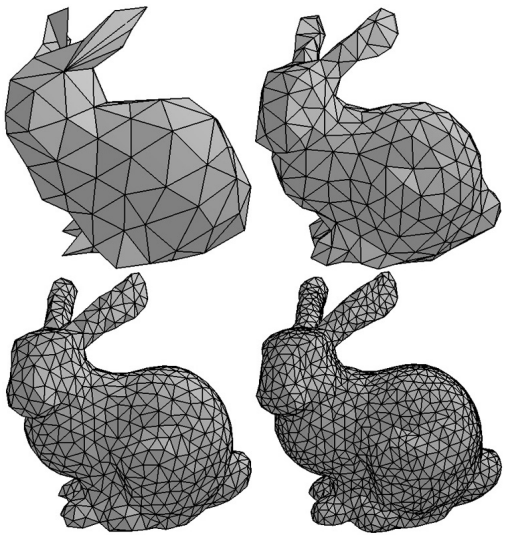


Figure 1: Example of some mesh models of the same object with different resolution

According to Chen et al. [5] active vision perception in robotics reached a peak in 1998 and became very active again in the last years due to the variety of applications it can handle.

Among them, inspection (15%) and object modeling (9%) are the most addressed tasks.

The planning of sensor views based on 3D sensor data is commonly referred to as Next-Best-View (NBV) planning [20]. NBV means that the robot needs to decide where to position the camera next in order to successfully scan the object. The NBV problem has been widely addressed since the 1980s but still remains an open problem. Scott et al. [20] gives a good overview of model-based and non-model-based NBV algorithms for object modeling, which can be volumetric or surface-based. In contrast to model-based approaches [14][21][24], where the views can be planned offline, for non-model-based algorithms [9][11][16], a next best view needs to be selected at runtime since no a priori information about the target object is given.

The difference between volumetric and surface-based algorithms lies on how the 3D sensor data is used: While surface-based algorithms relies on surface characteristics such as occlusion edges [19] or contours [22], volumetric algorithms use voxel occupancy information to model the scanning space and can be also used to obtain information for NBV [3][16][18].

Volumetric algorithms are widely used for modeling as they offers a compact way of space encoding. They can be also used for a coarse surface representation although they are not suited for high-precision modeling. Their principal disadvantage is the large memory requirement, but thanks to the higher computational power of the recent year's computers, some volumetric programs such as *Kinect Fusion*[18] have been arising to make 3D reconstruction technologies accessible to everyone.

If more accurate results are wanted, surface-based algorithms offer higher precision models than volumetric based ones. This firsts can also use a volumetric voxel grid, but only for collision avoidance and not for surface representation. Kriegel et al. [11][12] developed an improved occlusion edges surface algorithm, called *Boundary Search*, which offers very accurate results. With this approach, Kriegel implemented a trajectory NBV approach called Next-Best-Scan (NBS), that computes which is the best trajectory for a linear laser scanner to get the maximum information in one single movement.

Scott et al. [20] gives a general overview of the different view planning algorithms for object modelling, and designates the 3 basic elements that compose a general automated

3D reconstruction system: The imaging environment (Section 2.2), the reconstruction cycle (Section 2.3) and the view planning (Section 2.4). The imaging environment is made up of all the necessary physical components that allows the scanning stage, such as the sensors or the positioning system. The reconstruction cycle embraces all the software solutions to process the scanning data from the sensors and builds the virtual model. Finally, view planning is the process of determining the next viewpoints to place the sensor maximizing the useful information for the reconstruction. Each of these elements are further explained in the following sections, as well as their respective components.

2.2 Imaging Environment

In order to acquire a 3D model from a real object, the first step is to have all the necessary hardware to perform the operation. All this hardware and physical configuration builds the imaging environment (Figure 2), that is composed of:

- The target object from which we want to obtain the 3D model.
- Some fixtures to hold the object in the working space.
- A range camera to get the data from the target object.
- A positioning system to move the range camera and/or the target object in order to reach better quality viewpoints.

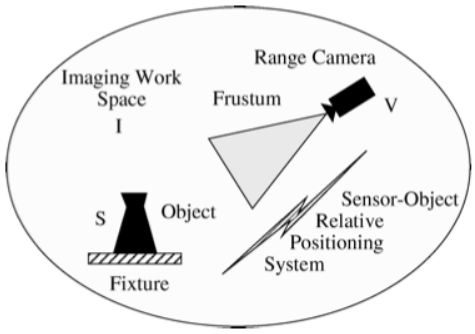


Figure 2: Imaging environment [20]

2.2.1 Range Camera

The basic component of the imaging environment is a range camera that acquires the 3D shape from the target object. There exists a lot of different technologies for measuring an object shape, but they can be categorized into active and passive devices [4].

Passive range-finding devices are the ones which obtain data from the environment without having any integrated illumination source. The most common passive devices are the stereo vision cameras, which get 3D shape data through two spaced sensors (Figure 3). Although they can provide accurate measurement at short stand-off distances, they depend on visual texture of the target object and are always subject to illumination constrains and interferences. In addition, they require a lot of computational power to process both camera images and identify the objects in them.

Figure 3: Stereo vision system, composed of two cameras

On the other hand, active sensors have an integrated illumination source so they are less susceptible to external interference. Active sensors can be divided into two main subcategories: Time-of-Flight (ToF) and triangulation. ToF sensors are often used for long range applications as they require very accurate timing sources and typically provide modest resolution. Triangulation sensors, in turn, are able to perform very precise and dense depth measurements at relatively close distances, so they are widely used in the 3D reconstruction field. They are based on the principle of triangulating a measurement spot on the object from a physically separated camera optical source and detector (Figure 4).

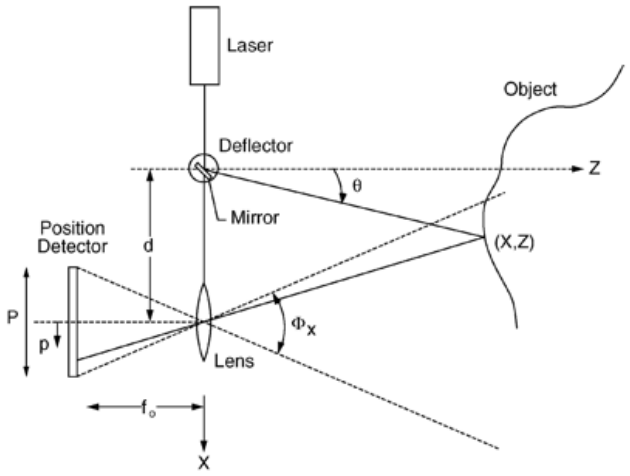


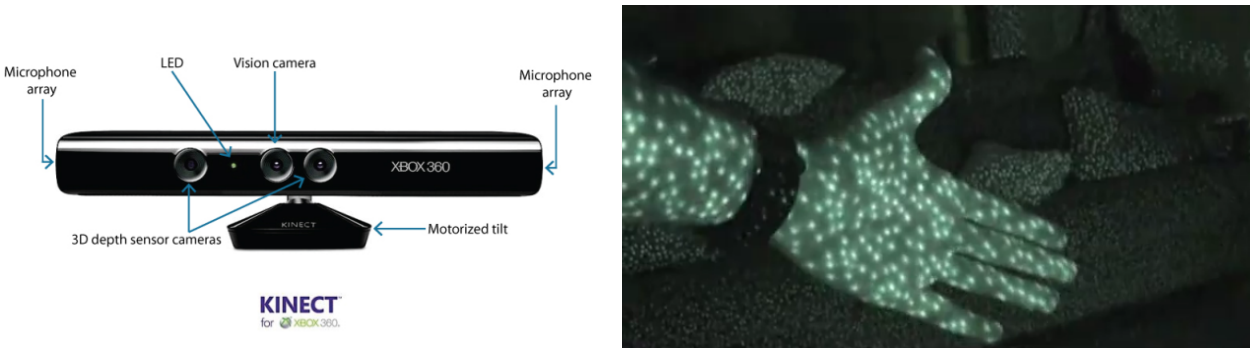
Figure 4: Conventional active triangulation [6]

2.2.1.1 Microsoft Kinect sensors

All the active depth sensors have been always very expensive, but everything changed when Microsoft launched the Kinect sensor in 2010. Microsoft Kinect was first created as a sensor for gaming purposes connected to the Xbox 360 console [25] (Figure 5a). However, due to its low cost, the platform opened a huge amount of possibilities in the 3D reconstruction field.

The first model of Kinect is based on triangulation, in a process called structured-light sensing [7]. The device projects a pseudo-random structured pattern produced by a near-infrared laser source that illuminates the entire field of view (Figure 5b). With this image, the camera then looks at small windows of the captured image and attempts to find the matching dot pattern in the projected pattern. However, this system has its own limitations: As it needs to identify the pattern it can not distinguish a single point without its neighbours, so thin objects are difficult to detect. Moreover, other external sources of infrared light like the sun makes the points undistinguishable, so the Kinect mostly fails outdoors.

In 2013 Microsoft launched the second version of Kinect, addressed to its new generation gaming console [26]. This new version changed completely the technology used for the first one: Instead of using an structured-light pattern, it implements a ToF active sensor. By the time it was launched, all the commercial ToF sensors in the market were very expensive and



(a) Kinect v1 components **(b)** Projected infrared pattern

Figure 5: Kinect v1

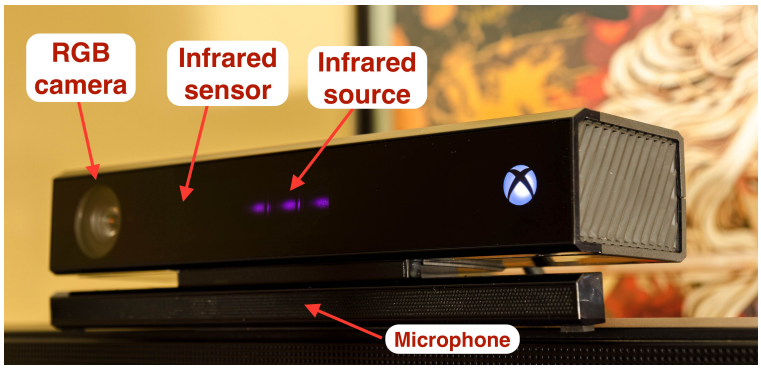


Figure 6: Kinect v2

with less resolution than Kinect v2. This new version has a wider field of view ($70^o \times 60^o$ vs $57^o \times 43^o$) and greater depth resolution (512×424 vs 320×240) than the first Kinect. This new version is also more robust against illumination changes, but it is still unable to get good results with strong infrared sources such as sun light. Both cameras operating ranges start from $0.5m$ up to $4.5m$, so their minimum range have to be taken into account for the positioning system.

The Microsoft Kinect v2 sensor relies upon a novel image sensor that indirectly measures the time it takes for pulses of laser light to travel back and forth from Kinect to a target surface. Each sensor pixel is divided in half: Half of this pixel is turned on and off in a high rate so it is absorbing or rejecting photons of laser light. The other half is doing the same but at 180 degrees out of phase, so there is only one half receiving photons at each time [27].

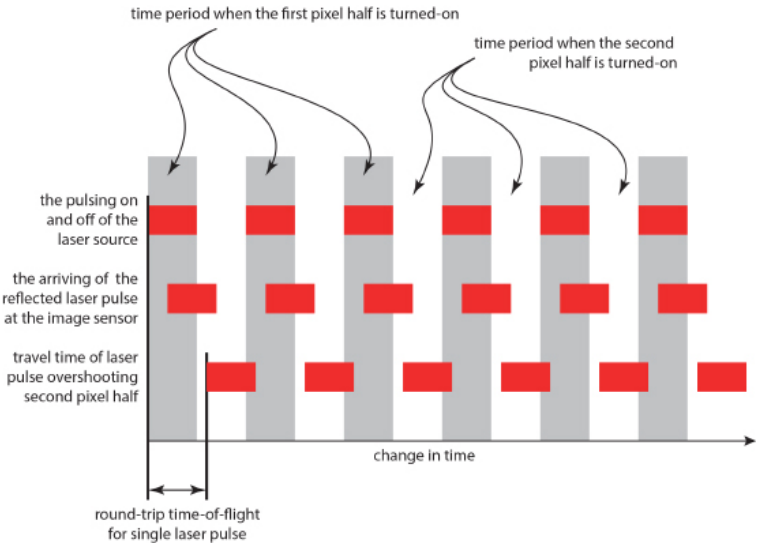


Figure 7: Kinect v2 Time of Flight working scheme [27]

As the laser light source is also being pulsed in phase with the first pixel half, the time offset between theses laser pulses and the two half pixels measures the distance to the target's surface (second row at Figure 7). This offset is measured through the relative amount of light each half pixel receive, so the higher ratio of photons the second half absorbs respect the first one, the farther the object is. However, if the object is farther than the measuring rate the obtained data will have some ambiguity (third row at Figure 7). For this reason, Kinect v2 takes two measurements, where the first measurement is a low resolution estimate with slower measuring rate, so there would not appear ambiguities in distance. The second measurement is then taken with higher precision using the first estimation.

Due to its low price and good performances, Kinect v1 and v2 have been selected as the most appropriate range cameras for the 3D reconstruction performed in this project. While Kinect v2 has better specifications than Kinect v1, ToF technology might not be the best one for 3D reconstruction so both Kinect models will be tested and compared.

2.2.2 Positioning System

A variety of viewing perspectives are needed in order to get images from all sides of an object. Thus, a positioning system is required to move the sensor, the object, or both.

In this project case, a KUKA LWR 4+ with 7 degrees of freedom [28] (Figure 8) will hold the Kinect sensor in its tool centre point, so in an eye-in-hand configuration. However, the minimum $0.5m$ range of the Kinect sensors has to be taken into account, so the robot has to be large enough to cover the different viewpoints keeping this minimum distance from the target object. The viewpoint space taken into account is an sphere with the centre on the target object and radius equal to the minimum sensor range specification.

Figure 8: KUKA robot [28]

Figure 9 shows the KUKA robot workspace. As it can be seen, a complete $0.5m$ radius view sphere can not be placed between the robot and its maximum range if the target object lies in the same plane as the robot's base. For this reason, not all the points from the viewpoint sphere will be accessible for this configuration, so some experimental tests will be needed in order to evaluate if this positioning system is good enough.

In the future case when the system is implemented into a mobile robot, some other aspects will need to be taken into consideration. The first one will be the position error, because odometry is not accurate enough and the error is accumulative. Other consideration will be the accessible viewpoint space, because the robot will usually need to go around the table

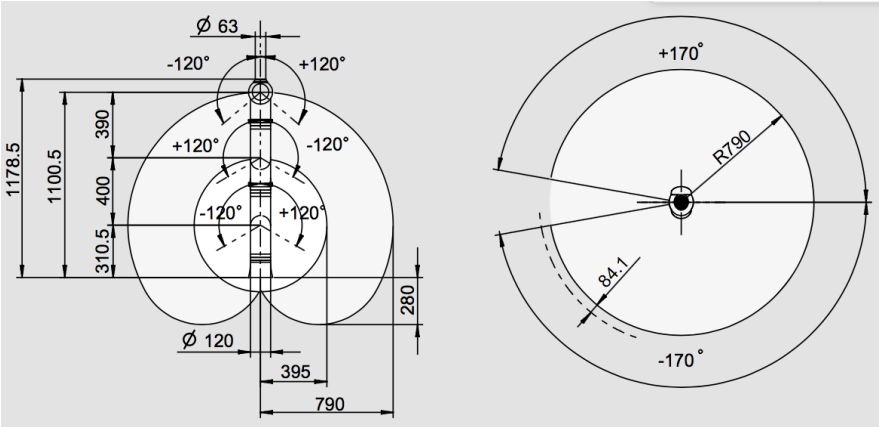


Figure 9: KUKA Workspace [28]

where the object is located in order to get all the views.

2.2.3 Target Object and Fixtures System

Finally, the indispensable part of the system is the target object from which its digital model is wanted to be reconstructed. The most important properties of an object in order to be reconstructed are its size, its surface shape and its materials.

The size affects both the range camera technology and the positioning system. By this way, a small object might need a more precise sensor whereas a bigger target maybe its not fitting in the positioning system.

Using the proposed system, a Kinect sensor mounted in the KUKA LWR 4+ hand, the target object to scan can not be too small due to the limited resolution of the sensor nor too big in order to reach all the necessary perspectives. For this reason, the chosen bounding box size for the target object is between 10 centimetres for the shortest edge and 25 centimetres for its longer one.

The shape of the object is another key factor to take into account. Targets with concave surfaces are always harder to scan because of the inner unseen areas for the sensor. These concavities are the main reason why a view planning system is needed instead of a fixed trajectory around the object: Its important to detect the views from where it is possible to see all the hidden inner surfaces.

Object's materials will affect sensor performance depending on the technology that it is using. Taking into account Kinect infrared sensor technologies, the worst materials to detect are the translucid ones (glass, transparent plastic) and the reflective ones (mirrors, brushed metals, shiny plastics...) because IR light is less reflected towards the sensor. For these reasons, the scanned objects will have low-reflecting opaque materials to ensure good sensing performance for Kinect cameras.

The scanning system will be located indoors in a room where the windows are covered to prevent sun light to interfere sensor's performance. For this reason, the environment is isolated from strong IR sources and it is illuminated only with artificial fluorescent light.

Regarding the fixtures system, the system is mounted in a fixed table inside the Laboratory in a similar configuration as Monica et al. [16][17]. If the robot is not large enough to scan the object, a solution would be to locate the object in a turning table [9] or hang it to the ceiling, so the robot would be more accessible to the viewing positions. Otherwise, the object can just be placed over the table so no extra fixtures would be needed.

2.3 Reconstruction Cycle: *Kinect Fusion*

The classic model building cycle consist of four main phases: plan, scan, register and integrate [20]. This cycle works as follows: Firstly, a view plan for reaching the next best views is computed from the 3D model. Then, the sensor is moved to the previously computed positions and scans the object. After that, acquired range images are registered by and image-based registration technique such as the Iterative Closest Point (ICP). However, if the positioning system is error-free this phase is not necessary. Finally, in the integration phase the registered images are combined in a single, non-redundant model. The view planning will use the integrated model for computing the next best view, and all the cycle is repeated again until some stopping criteria is reached.

From all the reconstruction cycle, there exists different solutions that implement the scan-register-integrate phases, but automated view planning still remains as an open problem. For this reason, in this sectionan accessible and robust scan-register-integrate solution (*Kinect*

Fusion [18]) is commented, and different view planning approaches are explained on next Section 2.4.

2.3.1 Original Operation

Kinect Fusion is an iterative algorithm for real-time tracking of a moving depth camera (registration phase through ICP) and 3D fusion of the environment observations in a volumetric data structure (integration) [18]. In order to reach real-time reconstruction, this software exploits highly parallel GPU technologies (*Nvidia CUDA*).

Kinect Fusion represents the environment as an implicit surface model using a Truncated Signed Distance Function (TSDF). This function maps the 3D coordinates into a signed value representing the distance to the nearest surface (negative inside the object and positive outside, look Figure 10). The environment is divided into a 3D voxel grid in which each voxel has the same size and contains two different values: the TSDF sampled value v between -1 and 1, and a weight w that counts the number of times this voxel has been observed.

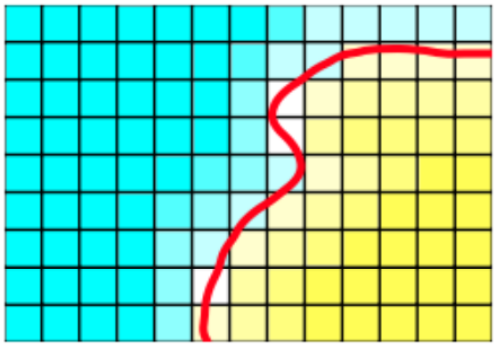


Figure 10: 2D TSDF volume representation [16]: the cyan area represents the positive values (outside) whereas the yellow area represents the negative ones (inside). The red line is the object surface (zero-crossing value)

The *Kinect Fusion* iterations are composed of four different steps:

1. *Ray casting*: A virtual sensor is placed in the last known position and a synthetic depth map (a pointcloud) is simulated from this view using the stored TSDF volume information. This synthetic depth map is obtained through the simulation of each ray

of the sensor though the TSDF volume: when the ray find a zero-crossing (which means that it crossed the object's surface) a new point is stored in this synthetic map. Each ray is evaluated in parallel thanks to the GPU *CUDA* architecture.

2. *Depth map conversion*: The depth image obtained from the real sensor and converted into an organized pointcloud.

3. *Camera Tracking*: The pointcloud obtained from the sensor is aligned with the synthetic depth map using a modified point-to-plane ICP algorithm. After that, the motion of the sensor can be estimated from the ICP transformation.

4. *Volumetric integration*: The pointcloud is converted into global coordinates and merged with the TSDF volume. The sample values v for each voxel are weighted taking the weight value w into account and updating it.

A limitation of *Kinect Fusion* is that it can not work with data outside the TSDF volume. Besides, this volume can not be expanded indefinitely due to limited GPU memory. For this reason, an improved version called *KinFu Large Scale* has been developed to handle large environments based on downloading part of the 3D representation from the GPU to the CPU RAM memory. However, this procedure called *shifting* losses the weight values w from the TSDF volume, so the accuracy is slightly reduced. However, the aim of this project is to get a model from an object that will be smaller than the TSDF volume, so this issue does not affect this work if the volume is correctly set.

2.3.2 Added Functionalities

In this project the *Kinect Fusion* algorithm is implemented in ROS [35] due to its accessibility, quality and integration with the Kinect sensor. The implemented version [16] is slightly modified in order to integrate useful tools for the view planning algorithms:

- It can disable the ICP tracking (which can induce to position errors) and perform it using a more accurate external position reference. By this way, if the sensor is placed

in the hand of the KUKA robot, the position of this hand (which is given by the robot, so it is very precise) can be used for the 3D reconstruction.

- It has a modified version of the ray casting algorithm which is very useful for view planning. This new version can be accessed at any time and it can simulate the view from a set of desired points, giving as an output the number of seen voxels and the number of unknown ones for each view. This operations have high computational costs, but like the original ray casting algorithm, this modified one also takes advantage of the parallel *CUDA* processing so it can be performed faster.

2.4 View Planning

The view planning problem consist in the search for the next viewpoints to place the range camera at every moment. However, in order to move the end effector to the different poses, the robot motion planning problem has to be solved in the first place to move the robot through a path without collisions.

Firstly, this robot motion planning problem is explained in the following Section 2.4.1. This is the planner who takes into account the robot and world models to move towards the targets, finding the best path in the joint space and avoiding collisions. This planning problem is handled by *MoveIt!*[29], a software with ROS integration which includes a lot of different functionalities related to robot motion planning.

After that subsection, two view planning algorithms are explained: Next-Best-View and Best-Path Planning. These planning algorithms use a defined set of reachable viewpoints, with some quality information attached to them, to determine the next one to go. The movement between to arbitrary viewpoints is then computed with the *MoveIt!* motion planning software to avoid collisions.

2.4.1 Robot Motion Planning Problem

The goal of robot motion planning is finding a collision-free trajectory that reaches the goal as fast as possible. The general problem of motion planning can be stated as follows: Given,

- A start pose of the robot.
- A desired goal pose.
- A geometric description of the robot.
- A geometric description of the world.

The objective is to find a path that moves the robot gradually from start to goal while never touching any obstacle [30].

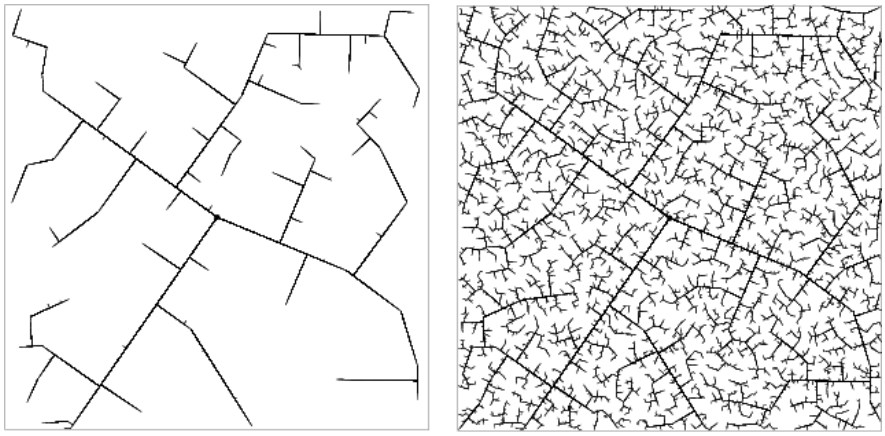


Figure 11: RRT graph in a 2 DoF configuration Space [30]

This problem can be solved through several methods, but due to the high dimensionality of the configuration space of the KUKA robot (7 DoF), the sampling-based bidirectional Rapidly Exploring Random Trees (RRT) method has been chosen. This method, also known as RRTConnect, is one of the most widely used because of its easy implementation and good balance between greedy search and exploration. As the world in our application is not very complex, this method is capable to find good solutions for all the possible movements, so other motion planning methods are out of the scope for this project.

The idea behind the RRT method [13] is to aggressively probe and explore the configuration space by expanding incrementally from an initial configuration q_I. The explored territory is marked by a tree rooted at q_I, and it will expand until reaching a near configuration from the final one (Figure 11).

RRT Connect is a more effective method, since it starts two RRT trees: one from the starting configuration q_I and another from the final one q_G, and after some iterations it tries to connect them. This modification is helpful to solve certain complex situations where an unidirectional RRT would fail, like in Figure 12.

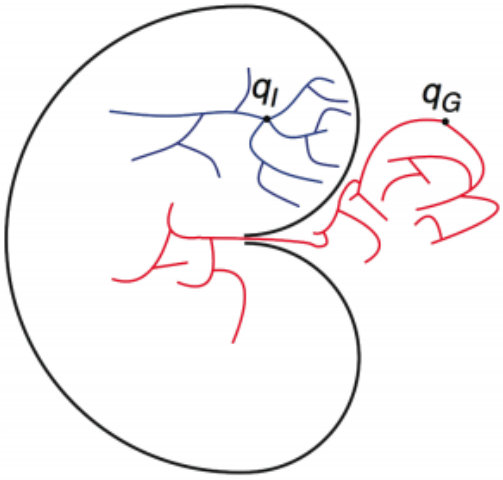


Figure 12: RRT Connect graph solving a complex environment (bug trap) [30]

2.4.2 Next-Best-View

Inside view planning, Next-Best-View (NBV) is one of the most used methods to select the next viewpoints to place the range camera. This method consist on assigning some viewing properties at each of the available viewpoints, and select the one with higher viewing quality.

This work is implementing the NBV algorithm from Monica et al.[16] with some modifications, and it will be compared to the developed new approach. For this reason, the integrated NBV algorithm takes advantage of the *Kinect Fusion* added functionalities: From each viewpoint, a view is simulated by the ray casting algorithm explained in Section 2.3.2,

so it has information of the known and unknown voxels. By this way, the NBV algorithm simply chooses the viewpoint with higher unknown voxels, and thus, the best one to explore.

This approach is very simple and can give good results, but it has some important drawbacks:

- It does not take into account the path to reach the next best view, so it might not look properly at the object during the movement.
- As it is always going to the global maximum, sometimes it can skip nearer local maximums and need to come back later on, losing a lot of time.

Even though surface-based scanning has better specifications (see Section 2.1) the implemented system will use only volumetric algorithms for the NBV, acquiring voxel information from the *Kinect Fusion* algorithm. Low-cost RGB-D cameras such as Kinect do not have enough precision and resolution to notice big differences between these algorithms, so as a first approach the volumetric one will be tested like in [16]. However, a surface-based scanning algorithm can be later adapted to the system like Monica et al. [17].

2.4.3 Best-Path Planning

This project has focused in implementing an improved view planning algorithm that overcomes the common NBV problems and offers better and faster results: The Best-Path (BP) planner. For this reason, instead of looking only to the single best view at each iteration, a trajectory approach has been implemented. This approach was mentioned in R. Monica as future work [16], so this thesis can be considered as the continuation of R. Monica's investigation in this specific topic.

However, before building a smooth trajectory it is essential to know which are the optimal zones to cross in order to minimize the total time and offer a good perspective of the object during all this path. That is why this project has not considered smooth global trajectories: Instead, the trajectories have been divided into path-points, so the robot is following a path going to viewpoints near from each other. By this way, as all the viewpoints are supposed

to be reachable and pointing to the object, the followed path will overcome the typical NBV drawbacks commented in Section 2.4.2.

To implement this viewpoint path the Dijkstra algorithm has been used [15], because it will try to minimize the total cost of the path that its following. This cost can be related to different parameters, such as the distance between points or the known/unknown voxels of the next path-point.

Three different behaviours related in how the Dijkstra path is computed are tested in this project: *Dijkstra Dynamic*, *Dijkstra NBV* and *Dijkstra Dynamic NBV*. Figure 13 shows a general example of how do they work, where the red path is the computed one by Dijkstra algorithm and the objective is to minimize the edges cost, which can vary at each step.

- *Dijkstra NBV*: Dijkstra path is computed with the global best view as its final target. During all the path points, Dijkstra is not computed again until the first target is reached.

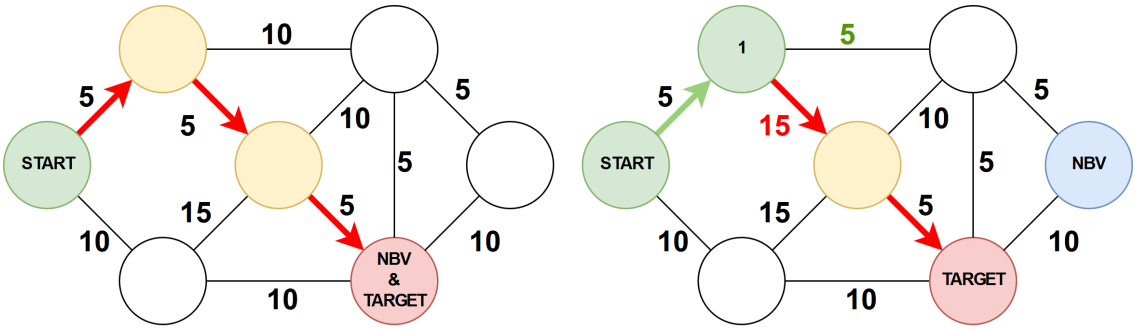
  This algorithm shows a first improvement from the original NBV: Even if this is following the exact same behaviour, as both go to the same objective, *Dijkstra NBV* assures that the path reaching it crosses the best intermediate views while original NBV does not. Nevertheless, as seen in Figure 13a, variations in the costs at runtime due to model updates might reduce its efficiency.

- *Dijkstra NBV Dynamic*: Like the previous behaviour, Dijkstra path is computed with the global best view as its final target. However, this one computes Dijkstra path again at every path point pointing to the first target until it is reached.

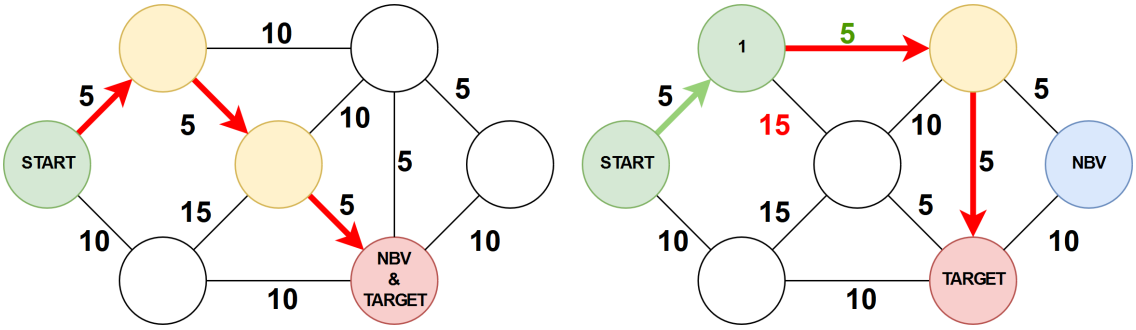
  This behaviour is thought to be more optimal than the first one at certain circumstances: When moving towards the target and the model is being updated, maybe some viewpoints in the path are no longer interesting so it is better to take a slightly different path to reach the same objective. As seen in Figure 13b, this algorithm can avoid bad moves due to cost variations but it still points towards the first target, which can cause it to lose efficiency when the global best view changes.

- *Dijkstra Dynamic*: Unlike the previous behaviours, this one does not keep the original target. For this reason, at every path point the updated best view is found and a completely new Dijkstra path is computed towards this new target.

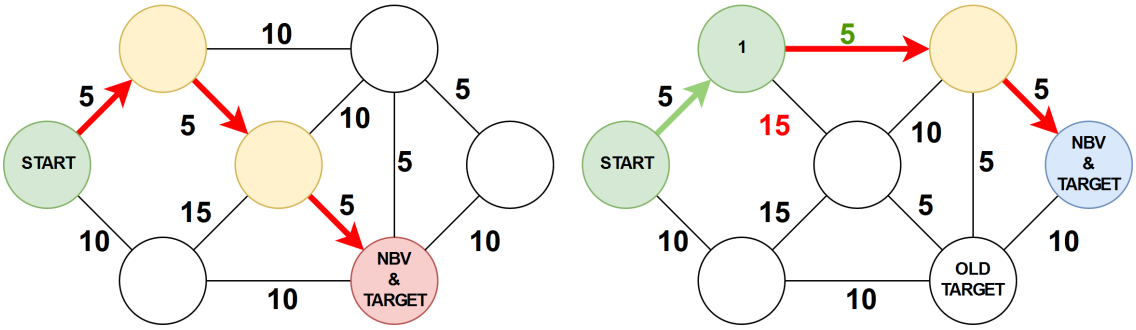
  This variation is focused to better explore local maximums near the starting point. This can optimize the path planning in some circumstances where the model is complex and a deeper exploration of the zone is needed. However, in other circumstances it can stay too much at this zones loosing time. Figure 13c shows a specific scenario where this algorithm is the most appropriate, but this situation might not happen frequently.



(a) *Dijkstra NBV* behaviour



(b) *Dijkstra Dynamic NBV* behaviour



(c) *Dijkstra Dynamic* behaviour

Figure 13: Different BP Planner behaviours in a cost-changing environment

3 System Implementation

In this section all the implementation process is explained. In the first Section 3.1, the used software and hardware environment are detailed to show which are the best configurations to make the system work. After defining the used system, general schemes of the software are shown in the following Section 3.2 to better explain all the program structure. After that, the software is explained in more detail in the Planner and MATLAB node subsections (3.3 and 3.4). Finally, the adaptation of the already build nodes is commented in 3.5.

This implementation project has been performed with the aim of creating an initial 3D reconstruction platform for the Aalto University Intelligent Robotics Laboratory. For this reason, all the code has been uploaded in the Aalto GitLab page [37] with its installation instructions to allow new users to easily use and understand this platform.

In addition, Appendix A in this project shows detailed information of the *rqt_reconfigure* window options that can also be a useful guide for who is performing a reconstruction. Appendix B is also useful for understanding the ROS structure, so both appendices are included in the GitLab repository. However, if greater detail about the system is required, this thesis will be also available for everybody who wants to improve this system.

3.1 System Environment

This system has been implemented in ROS Indigo under Ubuntu 14.04 LTS operating system. Although the *ros_planner* package, which is the core ROS package developed in this project, it is also working under newer software versions (tested in ROS Kinetic under Ubuntu 16.04 LTS), other dependencies and libraries might not work properly if they have not been updated in their original repositories. MATLAB scripts have been developed under R2017a, but they are backward compatible to earlier versions; the only requisite is having the *Robotic System Toolbox* to run ROS inside the MATLAB environment.

When implementing the system some compatibility problems needed to be solved:

- Kinect v2 needs to be connected to an USB 3.0 port due to the bigger bandwidth respect to the Kinect v1. However, not all USB 3.0 controllers are compatible. The used computer's motherboard has only *ASMedia* USB 3.0 controllers, which are known not to work with Kinect v2 and the *libfreenect2* drivers library. For this reason, a PCI-Express card with USB 3.0 needed to be bought (Figure 14). This card has 2 USB 3.0 ports and uses a *Renesas* 720202 controller, which was reported to work with Kinect v2.

- The ROS node wrapper to work with Kinect v2, *kinect_bridge*, does not work under *OpenCV 3*. At some point in the implementation of this project the system suddenly stopped working. The original *kinect_bridge* code does not specify which *OpenCV* version should be taken into account in the compilation, so the *OpenCV 3* was imported and thus, not working properly. To solve this problem, in the *CMakelists.txt* file the *OpenCV* version "2.4.8" needed to be specified.

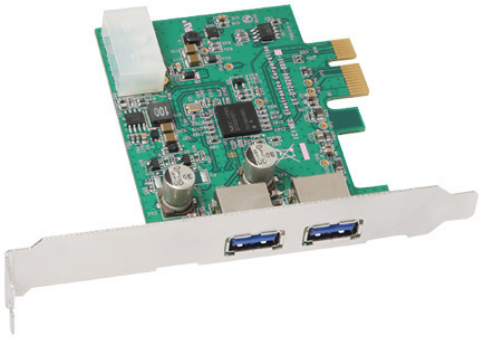


Figure 14: Sharkoon USB 3.0 Host Controller PCI-Express Card

3.2 System Structure

The system has been developed under ROS, but some functionalities have been added using MATLAB's *Robotic System Toolbox*. In this section the general structure of the system is explained, but in next Sections 3.3 and 3.4, ROS and MATLAB structures are explained in more detail.

First of all, the system functionality has to be defined. The aim of this system is to perform 3D reconstruction of an object using efficient planning methods. As commented in Section 2.4.2, the most used algorithm for this task is the Next-Best-View, so in this work more efficient methods using Dijkstra are used (Section 2.4.3) to find an optimized path to follow.

In Section 2.3 the reconstruction cycle is explained. It has 4 main phases: plan, scan, register and integrate. In the implemented system the scan, register and integrate phases are performed by the *Kinect Fusion* algorithms, so this work has been mainly focused in the planning phase.

The general procedure of this system to perform 3D reconstruction is the following:

1. The sensor camera (Microsoft Kinect v1 or v2) gives a pointcloud of the scanned model for each position (scan phase).

2. The *Kinect Fusion* algorithm processes the pointcloud and builds the 3D model (registering and integration phases).

3. The developed planning algorithms take into account the updated 3D model to determine the next path to follow in order to reach the best scanning positions (planning phase).

4. Finally the path to follow is given to the robot and it moves scanning all the path and performing this cycle again (scan phase).

3.2.1 Packages

The used ROS packages to perform all this operations are listed below. This package structure is shown in the scheme of Figure 15.

- *iai_kinect2* [36]: Necessary package for using Kinect v2 in ROS. This package links the *libfreenect2* library (which is the responsible to integrate the Kinect v2 system in Linux) to the ROS system, giving as an output several topics with all the data. This connection is done by the *kinect2_bridge* node, but other functionalities are included in this package such as a calibration suite o some simpler nodes for testing purposes.

- *ros_kinfu* [35]: Package which implements the *Kinect Fusion* algorithm in ROS with some added functionalities (Section 2.3.2). The main node is called *kinfu*, which is using the messages from the *kinfu_msgs* package to communicate with other nodes.

- *ros_kuka* [37]: Package that implements the KUKA LWR 4+ model and environment set-up (defined in *kuka_lwr*) to perform the motion planning algorithms using *MoveIt!* software. This package includes the *kuka_kinfu_pkg* node, which receives a pose and sends it to the integrated *MoveIt!* model to perform the real or simulated movement without collisions.

- *ros_planner* [37]: Original developed package for this work, which connects all the packages in order to reach the desired behaviour. This package includes the *planner* node, which gets the information of the 3D model from *kinfu*, communicates to the MATLAB node to get the best path to follow and sends it to the *kuka_kinfu_pkg* node to perform the robot movement. It can also configure the parameters from all the connected nodes with intuitive *rqt_reconfiguration* windows. This allows an easy way to configure the system and its behaviour while it is running. This node communicates to *kinfu* using *kinfu_msgs* and to MATLAB nodes through custom ROS services defined in the *planner_srv* package.

- *matlab_planner* [37]: Package with the original MATLAB scripts that performs the complementary functionalities that *ros_planner* need. The main *sphere_server_init.m* script starts the ROS service servers that will communicate with the *planner* node clients. These scripts use several ROS custom messages, so MATLAB needs to generate them from the ROS message packages. In *matlab_msgs* folder there are all the needed MATLAB generated files to use these custom messages.

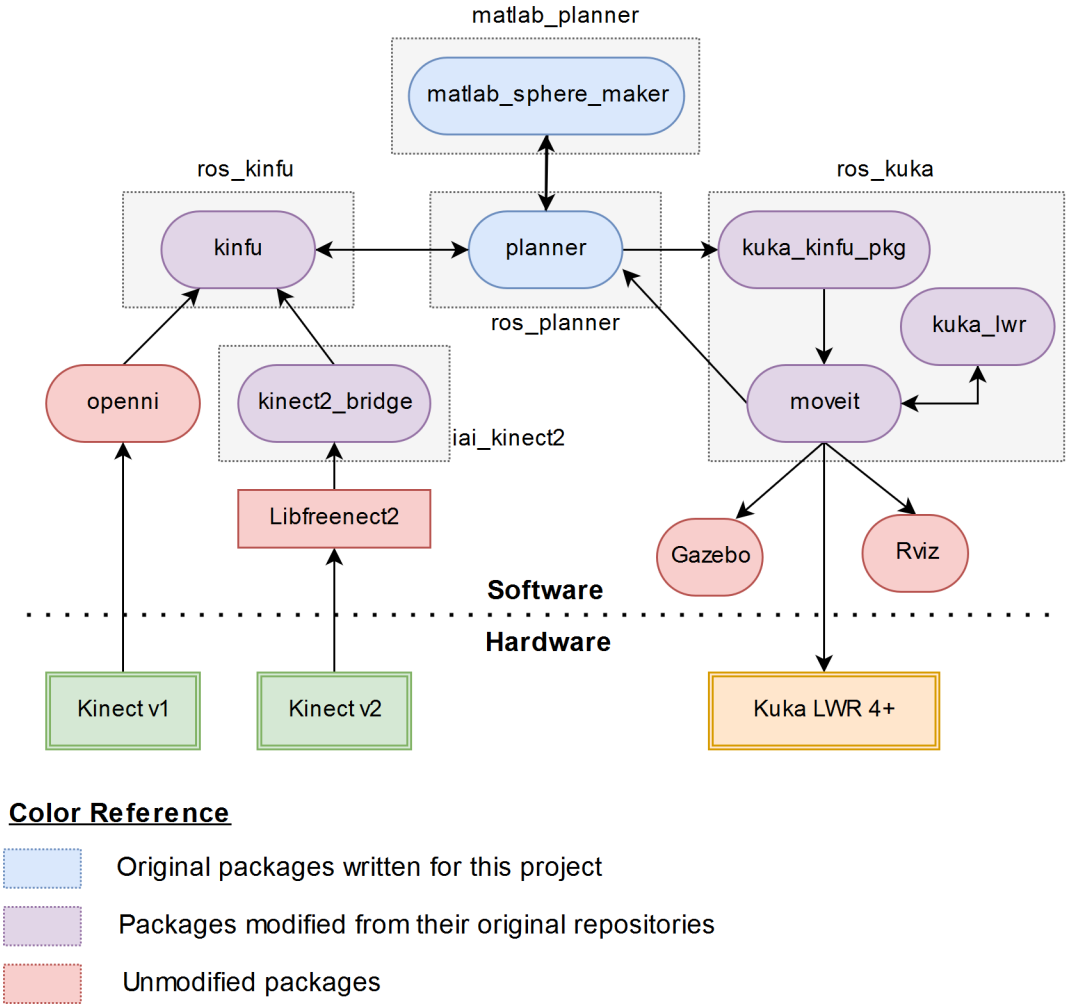


Figure 15: System scheme of all the packages used in this project

3.2.2 Topics

The communication between the different nodes is held by ROS topics using custom messages. Figure 16 shows a scheme of the topics used by the *planner* node.

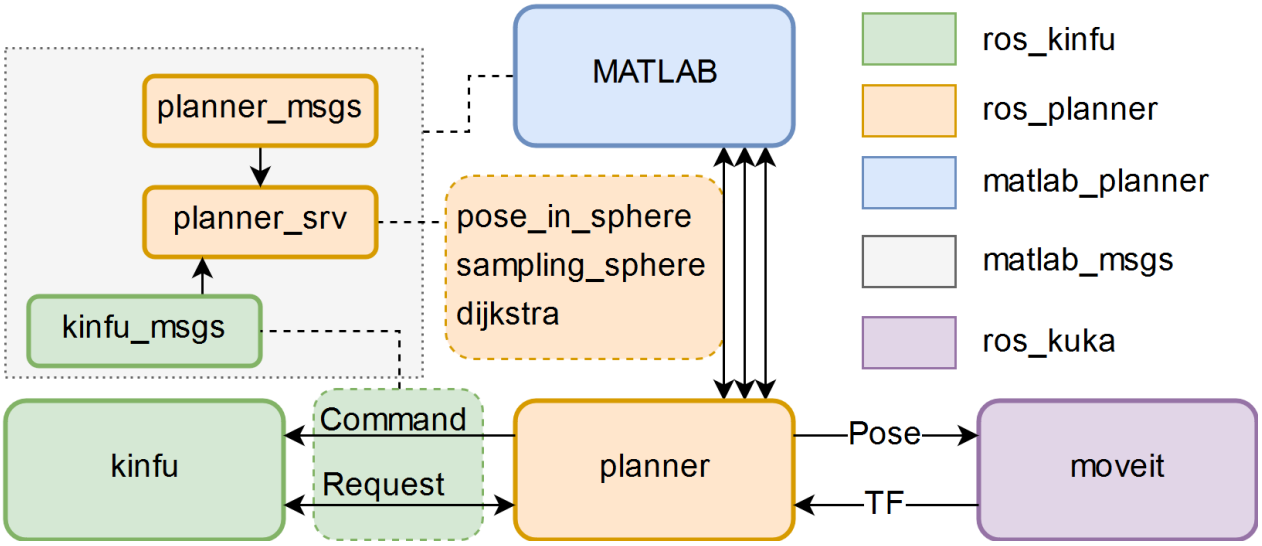


Figure 16: ROS topics scheme

Planner* communication with *kinfu

This *kinfu* node has different ways to communicate with the rest of the system: through commands or with requests. All the messages are defined inside the *kinfu_msgs* package.

Commands are used to change the *kinfu* behavior at runtime and are executed by sending a *kinfu_msgs/KinfuCommand* message to the topic *"/kinfu_command_topic"*. In this specific configuration, the used commands of the application are:

- *COMMAND_TYPE_RESUME* and *COMMAND_TYPE_SUSPEND*: To start or stop the *kinfu* operation. This is useful to pause the reconstruction phase when the system is being reset. Another utility is found when the continuous reconstruction is not wanted, so the algorithm has to be paused during the movement and resumed once the sensor is in the desired position.

- *COMMAND_TYPE_SET_FORCED_TF_FRAMES*: This command allows to choose between using the Iterative Closest Point (ICP) tracking algorithm or to force the tracked pose to stick to a TF frame and thus, disabling the internal ICP tracking.

On the other hand, requests asks *kinfu* to publish parts of the internal representation, optionally processed in a few ways. These requests have been implemented by actions (*actionlib*), so they have a request and response mechanism defined in *kinfu_msgs/Request.action* message file. The necessary requests for the reconstruction are the following:

- *REQUEST_TYPE_PING*: This request is useful to test the *kinfu* communication. In addition, used among the *request_reset* parameter, it erases the current 3D model and restarts the algorithm.

- *REQUEST_TYPE_GET_MESH*: This request gives a *pcl_msgs/PolygonMesh* message of the current 3D model as an output. In this application, the mesh is saved in a *.vtk* file.

- *REQUEST_TYPE_GET_CLOUD*: This request gives a *sensor_msgs/PointCloud2* message of the current 3D model as an output. As the output of this application is a mesh (it holds more information), this pointcloud is only used for visualization purposes.

- *REQUEST_TYPE_GET_VOXEL_COUNT*: This request gives as output an array (*std_msgs/UInt64MultiArray*) showing the number of known and unknown voxels seen by a specific viewpoint. If more than one viewpoint is specified, it returns the number of known / unknown voxels for all the views. This request is essential for the application, because this output is used to compute the next best view or the next path to follow by the robot.

Planner communication with MATLAB node

Inside *ros_planner* there is the main *planner* package and two message packages to perform the communication with MATLAB: *planner_msgs* and *planner_srv*. Those package messages were needed to be defined separately because MATLAB required it to import the custom ROS messages.

The communication between the *planner* node and MATLAB is executed through ROS services defined in *planner_srv*. The *planner_msgs* package only contains some messages required by this services. There are three main functionalities performed in MATLAB:

- *pose_in_sphere*: Given a certain sphere and three orientation angles, this service returns a pose for the camera in the defined sphere surface looking towards the centre. Although this service is not needed in the final application, it is very useful for testing purposes.

- *sphere_sampling*: This service returns an uniform distribution of viewpoints given a certain sphere. It also takes into account the current robot workspace, so the unreachable points are not included.

- *dijkstra*: Given the lists of known and unknown voxels, this service returns the best path to follow from the current position towards the next best view, using the Dijkstra algorithm.

Planner communication with *ros_kuka* nodes

Once the planner knows the next pose to go, it communicates with *kuka_kinfu_pkg* using a *geometry_msgs/Pose* through the *"/kuka_pose"* topic. Then, *MoveIt!* planner finds a suitable trajectory to reach this pose using RRTConnect, and when the motion planner finds it, the trajectory is executed on the real robot (or in the *Gazebo* simulation). In order to have feedback from the robot, the *planner* node checks the TF of the end effector link from the *MoveIt!* robot model, which is constantly updated as it moves.

3.3 Planner Node

The *planner* node is the core of the implemented system. Its duty is to connect all the different ROS nodes to perform the required tasks, implementing an efficient path planning for the reconstruction problem and showing an user friendly graphical interface to coordinate all the possible operations.

In last Section 3.2, the communication of the *planner* node with other nodes is explained. This section is taking a closer look to the node functionality, explaining how it works, which types of functionalities is able to handle and how its internally structured.

3.3.1 Structure and Classes

The planner node has been divided in several classes, each one in a different source file in order to maintain a clear and understandable code. By this way, the different code files are not very large and the main file is kept clean with only the essentials.

The scheme of the used classes is shown in Figure 18. In this figure each node has its own files named with the class' name: a header (*.h*) in the *include* folder and the source code (*.cpp*) in the *src* folder. In addition, some of this classes can be accessed from an user interface to change its parameters using the ROS *dynamic reconfigure* option. Necessary files for this functionality are located inside the *cfg* folder.

The structure is the following: There are four classes which handle the communication with the exterior: *Kinfu_Command*, *Kinfu_Request*, *Kuka_Op* and *View_Points*. This classes connect to a *dynamic reconfigure* server, so its behaviour can be modified in real-time. Then, a struct called *class_holder* has been made to include all these 4 classes in order to easily access them.

The communication of this nodes is explained in Figure 17, and it can be summed up as:

- *Kinfu_Command*: This class is responsible of sending any of the available commands to *kinfu* node to change its behaviour at runtime.

- *Kinfu_Request*: This class sends a certain request to *kinfu* node. Each request is an action that asks certain values of the reconstructed model.

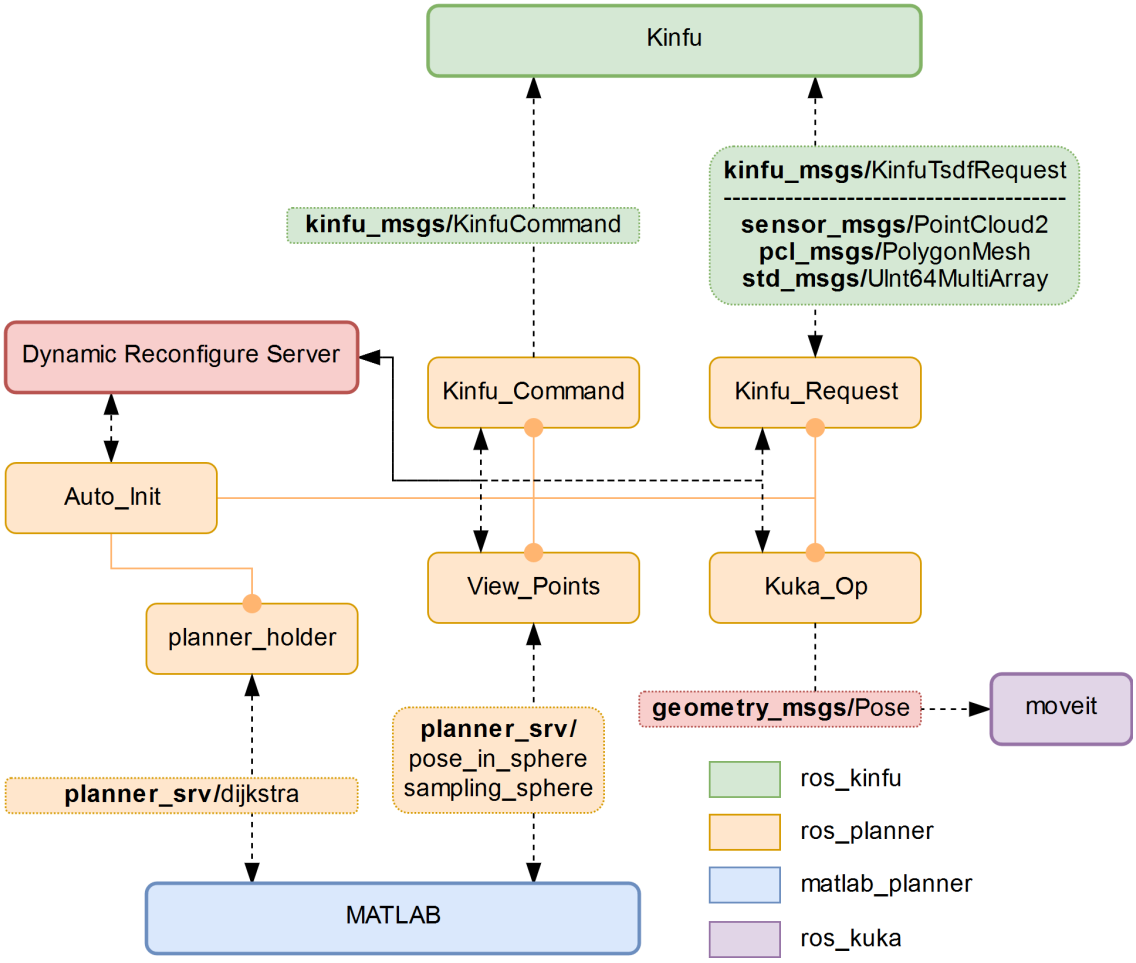


Figure 17: *Planner* communication scheme

- *View_Points*: This class is responsible to obtain the viewpoints for all the different applications of this program. It communicates with MATLAB and receives an uniform viewpoint sphere which can be configured through the user interface.

- *Kuka_Op*: This class communicates with the KUKA robot. It sends the necessary *geometry_msgs/Pose* message to the *MoveIt!* node inside *ros_kuka* in order to move the robot arm. It can send an arbitrary pose or a certain viewpoint of the generated set in the *View_Points* class.

To store all the configuration parameters from the user interface as well as the variables from the *kinfu* request or MATLAB responses, the *config_holder* class is defined. This class

holds all the variables and functions which are necessary for the rest of classes. The object of this class is defined as a pointer that is given and stored in the other objects when they are initialized. By this way, all the objects have the same pointer to the *config_holder* object, so they can access to all the necessary variables stored from other classes (without accessing at the same time). The *config_holder* class also has a *planner_tools* object, which handles a lot of different functions related to message conversions and TF transformations.

On the other hand, the *planner_holder* class includes all the functionalities related to the view planning layer. This class accesses to the known / unknown voxels count of all the viewpoints obtained through a *kinfu* request. After that, it communicates with MATLAB asking the Dijkstra path and decides which pose has to be sent to the real robot, sending it using the *Kuka_Op* class.

When the main code is executed, dynamic memory allocated objects from *config_holder*, *planner_holder* and *class_holder* are created, defined as pointers. By this way, it is easy to share this pointers through all the program pointing to the same objects and thus, the same shared variables.

Finally, a higher level class *Auto_Init* has been defined to help the user performing the reconstruction tasks. This class, which is also linked to the dynamic reconfigure server, shows an easy interface to perform the higher level tasks executing all the required actions automatically with the correct timing. For this reason, *Auto_Init* class controls all the *planner* classes and allows an easy operation through the *rqt_reconfigure* interface executing pre-defined tasks.

3.3.2 Graphical User Interface and Functionalities

As it has been explained in the last subsections, the *planner* node can be configured through a graphical user interface, which enables the user to perform different tasks in the runtime. This user interface is held by the *rqt_reconfigure* node, which is able to change the parameters from the nodes that are communicating with the *dynamic reconfigure* server.

In order to keep the parameters organized, five different classes with independent *dynamic reconfigure* communication have been defined: *Kinfu_Command*, *Kinfu_Request*, *Kuka_Op*,

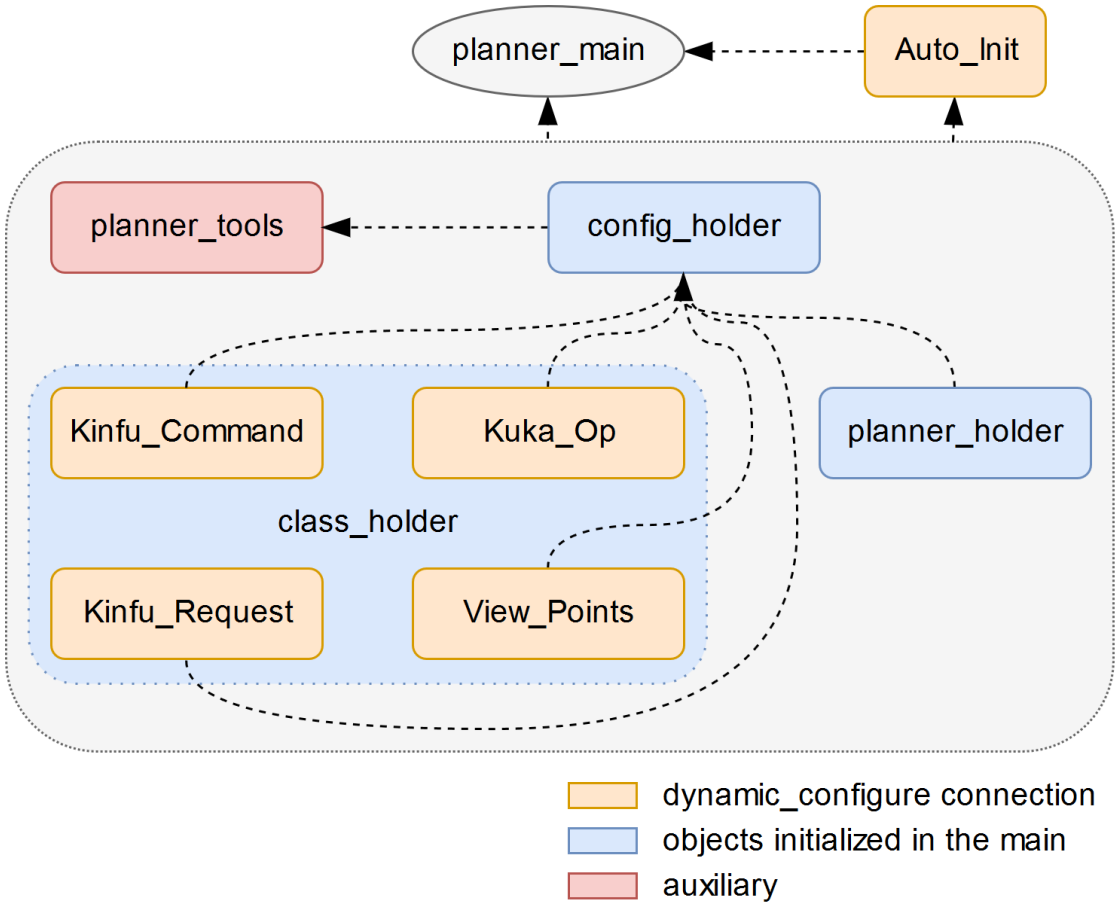


Figure 18: *Planner* class scheme

View_Points and *Auto_Init*. By this way, in the graphical user interface all the available parameters are grouped depending on their functionality.

In the Annex A there is the complete reference of the graphical user interface, showing in detail how do all the specific functionalities work. In this section just the *Auto_Init* ones are explained.

Figure 19 shows the *rqt_reconfigure* interface, in this case the one related to the *Auto_Init* class. In the left window the other classes configurations can be selected, so all the parameters are grouped leaving a clean interface. This user interface offers different interaction boxes depending on the type of parameter. Here we can see some check boxes for the boolean parameters, a slider to select an integer number and a drop-down menu to choose between

different options.

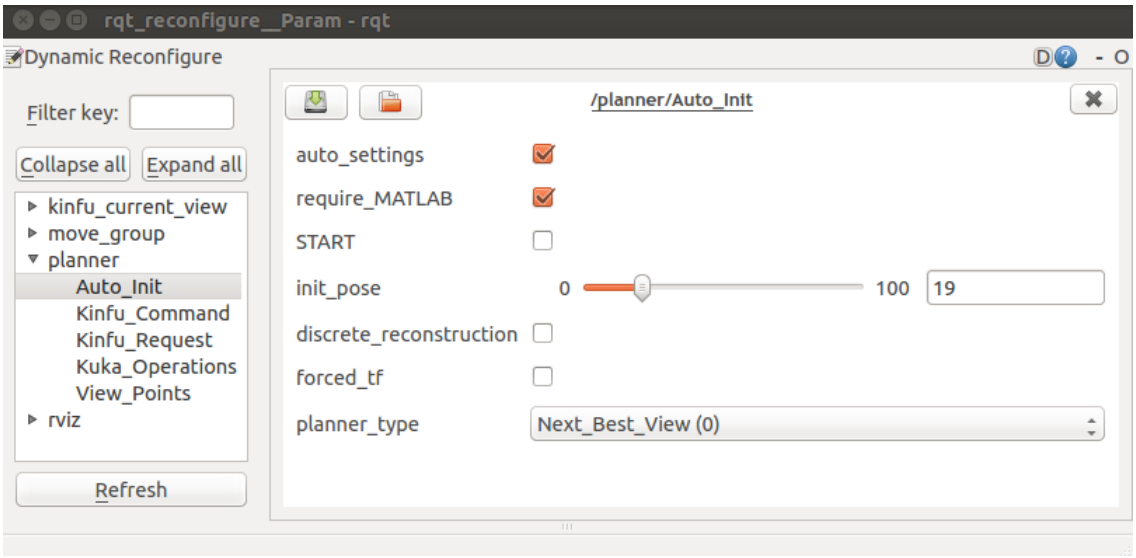


Figure 19: *Auto_Init rqt_reconfigure* window

As it is commented in Section 3.3.1, the *Auto_Init* class automates the procedure to perform the reconstruction tasks. The different configuration options of this class are:

- *auto_settings (bool)*: If *True*, the predefined configurations of the *Auto_Init* class are taken into account. If *False*, this class does not have any effect and the tasks need to be performed manually with the other classes configurations.

- *require_MATLAB (bool)*: If *True*, the program will be blocked until the MATLAB services are available.

- *START (bool)*: If *True*, the reconstruction task will begin. If *False*, the KUKA robot will go to the initial pose defined in the *init_pose* parameter, and reset the current 3D model.

- *init_pose (int)*: Slider to select the initial pose number to start the reconstruction. The pose numbers are determined by the uniform sphere returned by the MATLAB service, that can be visualized in the *Rviz* interface.

- *discrete_reconstruction (bool)*: If *True*, the reconstruction is performed just in the viewing points, so the *kinfu* node is deactivated in the translations and activated again when the view point is reached. If *False*, the reconstruction is performed all the time.

- *forced_tf (bool)*: If *True*, the *kinfu* reconstruction reference is forced to stick to the Kinect frame given by the *MoveIt!* node, deactivating the ICP algorithm. If *False*, the reference uses ICP.

- *planner_type (int)*: This drop-down menu let you choose between different planning algorithms in order to test their performance (explained in Section 2.4.2 and 2.4.3):

  - *Next_Best_View (0)*: Goes directly to the next best view.
  - *Dijkstra_Dynamic (1)*: Asks the Dijkstra path to reach the next best view. This algorithm goes to the next viewpoint of the received Dijkstra path, and when reaches it, computes the Dijkstra path again pointing to the updated next best view.
  - *Dijkstra_NBV (2)*: Asks the Dijkstra path to reach the next best view, and does not compute Dijkstra again until it finishes this path.
  - *Dijkstra_NBV_Dynamic (3)*: Asks the Dijkstra path to reach the next best view. This algorithm goes to the next viewpoint of the received Dijkstra path, and when reaches it, computes the Dijkstra path again pointing to the same old objective.

3.4 MATLAB Node

The MATLAB node is the responsible of performing some pose operations and the Dijkstra computation to find the best path. Although all these operations can also be implemented in C++ in a ROS node, MATLAB gives a lot of tools to easily perform geometric operations and offers good debugging tools that makes it very powerful for prototyping. Instead of translating the implemented code into C++ or Python, the direct MATLAB-ROS communication was a good alternative to test.

The MATLAB code generates three different service servers, which uses custom ROS messages. The necessary files to import these custom messages are located in the *matlab_msgs* package, that can be auto-generated with the ROS messages packages. Due to some MATLAB issues generating these files the messages and services packages need to be in separated packages: this is why the *planner_msgs* and *planner_srv* packages are not together. This files are provided in the Aalto Gitlab [37] for commodity, although the procedure to generate new messages is also explained there.

3.4.1 Services

***pose_in_sphere*:** This first created MATLAB service was thought to be useful for testing. It receives a sphere message (which includes its centre location and radius) and three angle values referred to Roll, Pitch and Yaw. MATLAB then computes a viewpoint in the surface of this sphere pointing towards the centre, located in the position defined by the asked angles. This pose is returned to the ROS node as a *kinfu_msgs/KinfuPose* message. This service is requested by the *View_Points* class, and can be asked through its *rqt_reconfigure* interface.

***sphere_sampling*:** This service returns a set of viewpoints where the Kinect can be located to scan the object. This viewpoints lay in a certain sphere and are distributed in an uniform way. However, this sphere might not hold the best positions to locate a real robot due to their workspace limitations. For this reason, these viewpoints are moved away to lay in a reachable position for the robot arm. After creating the viewpoints, the function creates a graph connecting them and stores it to be later used by the *dijkstra* service.

sphere_sampling receives a *sphere* message for locating the viewing sphere and an integer number related to the viewpoint density. In addition, other spheres are defined in the message to establish the workspace limits. As a result, this service returns a list of *kinfu_msgs/KinfuPose* messages where each element is a viewpoint, with their location and orientation.

This service is also requested by the *View_Points* class, so can be asked through its

rqt_reconfigure interface or from other classes, like *Auto_Init*.

***dijkstra*:** This service uses the previously stored graph that connects the nearby viewpoints, and uses the Dijkstra algorithm to find the best past to follow in this graph. The cost function takes into account the number of known and unknown voxels seen from each viewpoint. By this way, the output path will go through the worst views while approaching the objective.

dijkstra service takes as input the starting and ending viewpoint index, as well as the integer lists with the voxel information. As an output it return an integer list with the path and a value of the total cost.

This service is requested by the *planner_holder* class, so it can not be directly activated but indirectly through the *Kuka_Op* and *Auto_Init* classes when performing Best Path planning.

3.4.2 Uniform Sphere Sampling

In order to obtain points at a certain distance from the reconstruction object (Point of Interest), a viewpoint sphere has to be defined with the camera minimum's working distance as a radius. This sphere is defined as in Figure 20, so with $\varphi \in [-\pi, \pi]$, $\theta \in [0, \pi]$, $\psi \in [0, 2\pi]$, a viewpoint is defined in the sphere surface.

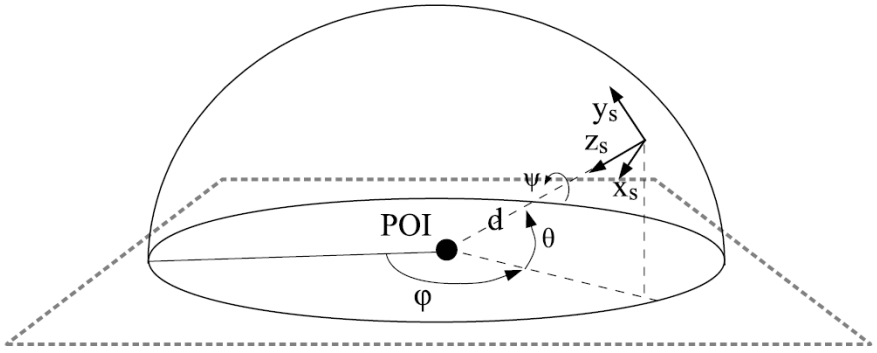
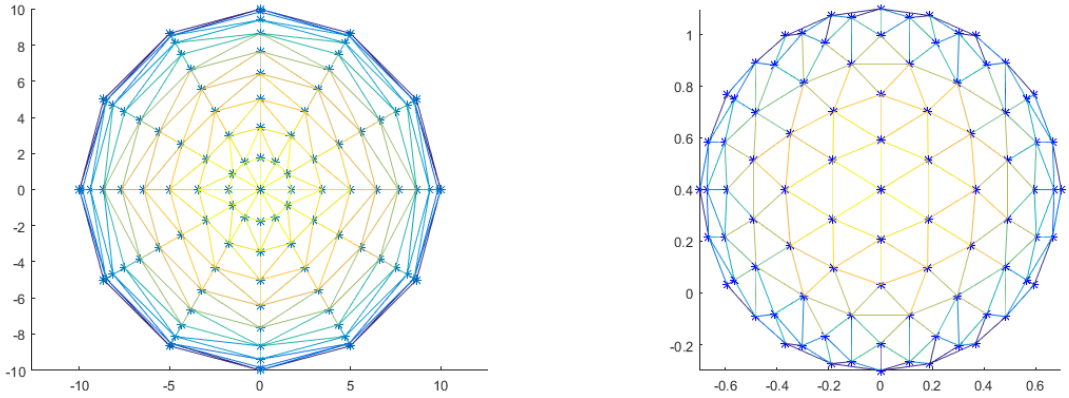


Figure 20: Sphere sampled around the Point Of Interest (POI) with radius d

R. Monica's work [16] defined the viewpoint sphere setting a constant angular step while assigning the points. Although this is a very simple method to implement, the uniformity of



(a) Sphere sampling with constant angle step variation [16]

(b) Uniform sphere sampling with a geometrical approach [31]

Figure 21: Sampled spheres, view from above

the viewing sphere is quite low because the points get closer with each other as θ increases (Figure 21a). In this case, when evaluating the viewing quality of the different areas the upper part takes more relevance because it has more points, so this approach is not optimal.

To improve this method, other sphere sampling methods were evaluated. A simple approach to overcome the uniformity problems is to decrease the point sample density as the θ increases. However, the best method is to use an uniform sphere sampling algorithm. There exists different approaches, but none of them reaches absolute uniformity. Due to its fast performance creating a large number of points, the geometrical approach described by Nick A. Teanby [31] is taken into account (Figure 21b).

3.4.3 Viewpoint Orientation

After creating the uniformly sampled points, next step is finding the correct orientation for each one. This orientation is defined as a 3×3 rotation matrix in the *kinfu_msgs/KinfuPose* message. The final orientation of the camera is shown in Figure 22, where the Z axis is always pointing towards the reconstruction object.

First of all, a rotation matrix that orientates the coordinate frame of the camera towards the scanned object has to be found. This means orientating the Z axis to cross the sphere

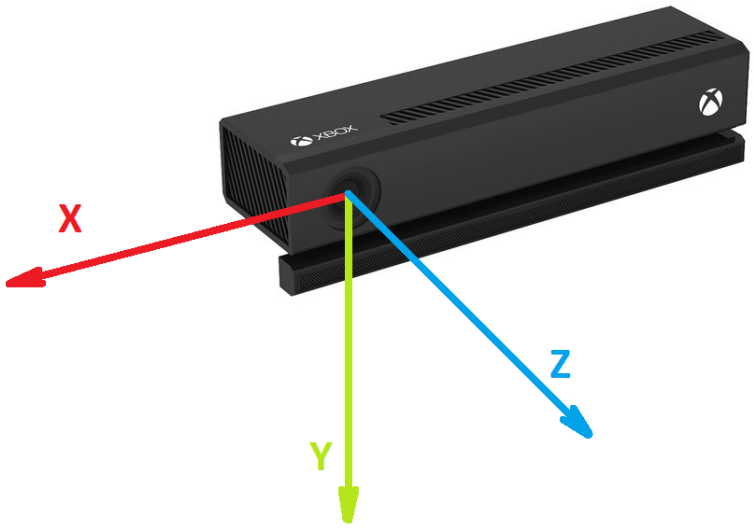


Figure 22: Kinect v2 coordinate reference frame

centre point. Starting from the world reference frame, the transformation that orientates the camera is the following: Operating in the local axis, first the frame has to rotate $[\varphi]$ in the Z axis, and after that rotate $[-\frac{\pi}{2} - \theta]$ in the Y axis:

$$R = rot_z(\varphi) \cdot rot_y(-\frac{\pi}{2} - \theta) = \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) & 0 \\ \sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(-\frac{\pi}{2} - \theta) & 0 & \sin(-\frac{\pi}{2} - \theta) \\ 0 & 1 & 0 \\ -\sin(-\frac{\pi}{2} - \theta) & 0 & \cos(-\frac{\pi}{2} - \theta) \end{bmatrix}$$

However, this equation is only valid to orientate the Z axis. Once with this orientation a certain angle has to be applied to rotate the view. Even though the reconstruction algorithm can handle any rotation when pointing towards the objective, it is necessary to have a good orientation to allow the robot arm reaching the viewpoint: For the same point only some orientations will allow the robot to reach the required pose. For this reason, a specific orientation rule has been applied to all the points: The camera will be upside down in the lower parts of the sphere and horizontal in the top (Figure 23). The middle points between the bottom and top will have a proportional rotation to allow a smooth transition.

This approach gives an intuitive positioning of the camera with natural transitions. However, there exists a critical area where the transition needs to rotate π radians from

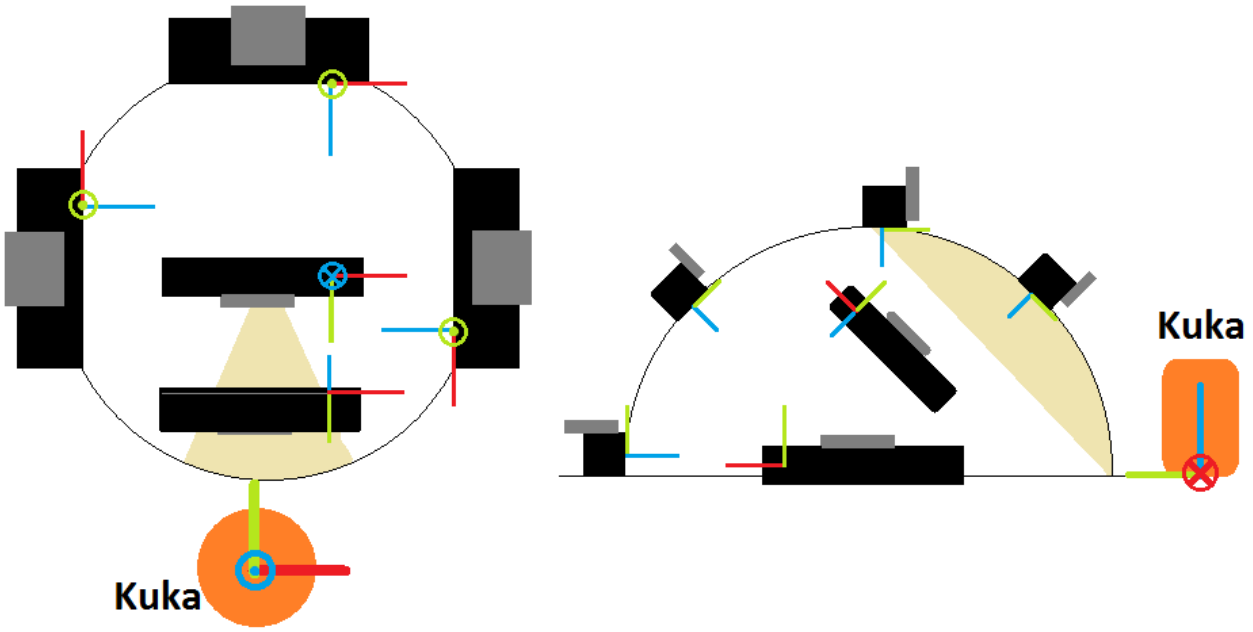


Figure 23: Scheme of the wanted orientation for the Kinect

bottom to the top. In addition, this zone is dangerous to cross by the robot arm due to the joint limits. For this reason, to cross it the robot will pass through the centre top point avoiding it. Then, the viewpoints of this zone will keep the same orientation as the top one. This approach can be defined as in the next equation, once the orientation has been previously applied with the previous one:

$$M = R \cdot rot_z(-\frac{\pi}{2} + \alpha) = \begin{bmatrix} \cos(-\frac{\pi}{2} + \alpha) & -\sin(-\frac{\pi}{2} + \alpha) & 0 \\ \sin(-\frac{\pi}{2} + \alpha) & \cos(-\frac{\pi}{2} + \alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where:

$$\alpha = \begin{cases} (\varphi - \pi) \cdot \dfrac{2\theta}{\pi} & \text{if } \varphi \in (0, \pi] \\ (\varphi + \pi) \cdot \dfrac{2\theta}{\pi} & \text{if } \varphi \in (0, -\pi] \\ 0 & \text{if } \varphi = 0 \end{cases}$$

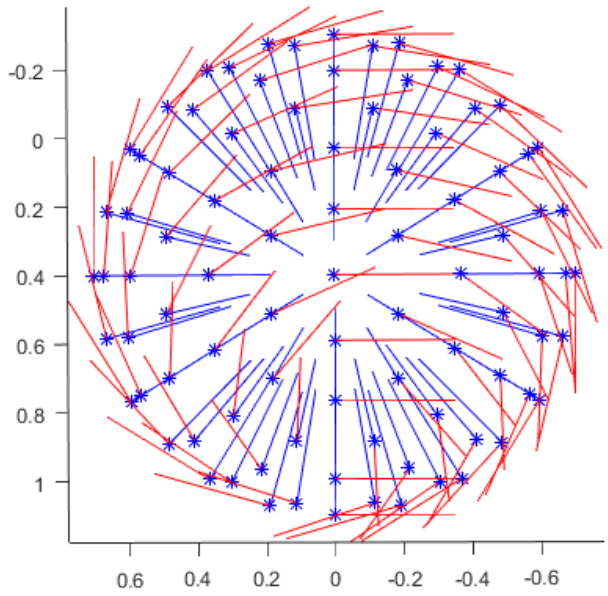


Figure 24: MATLAB output of the custom orientations (X and Z axis)

3.4.4 Workspace Adaptation

Even if all the viewpoints are correctly defined with their location in the sphere and their orientation, some of them will not be accessible for the robot. For this reason, all viewpoints need to be adapted to the KUKA robot workspace.

This workspace is defined with three spheres, shown in Figure 25:

- *workspace_sphere* ($r = 0.79$): The workspace sphere where the robot is able to reach the viewpoints. It is centred in the robot's second joint, and all the viewpoints out of this sphere are discarded.

- *limit_sphere* ($r = 1.5$): Only the points inside this limit sphere are taken into account. This is used as a method to model the table fixture limit: the centre of the sphere is located above the object position, so only the points above the table are taken. This is a versatile method to define the table, because it can easily model other fixtures with any orientation.

- *out_sphere* ($r = 0.4$): Sphere to exclude points, so all the viewpoints inside it are discarded. This is used to define the robot's inner workspace where it can not access due to its own joint limits.

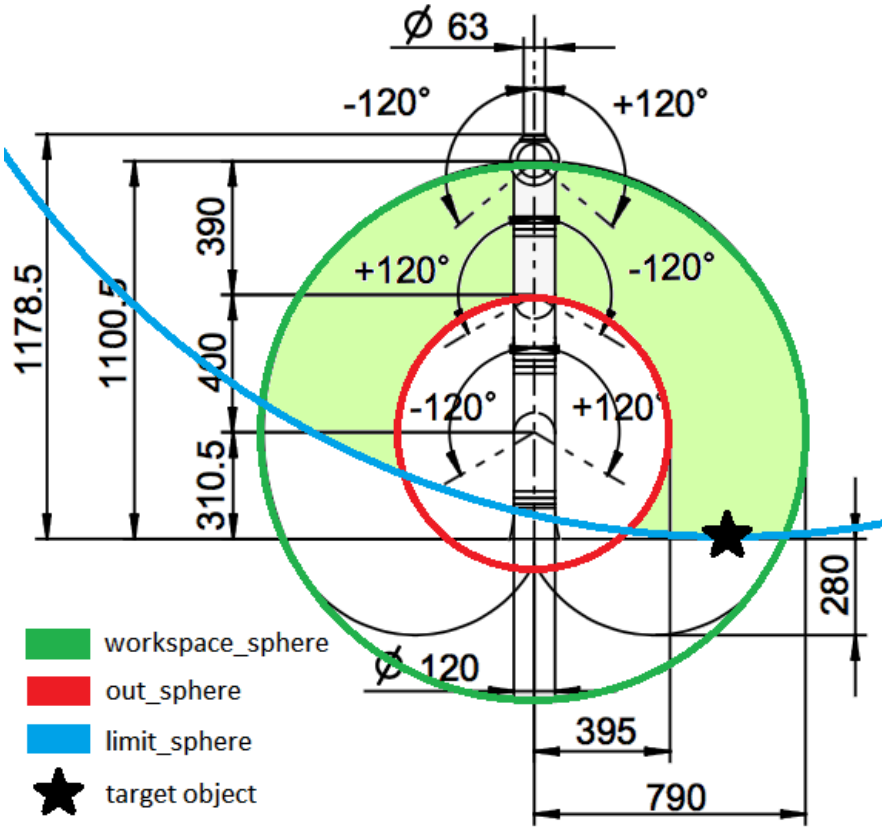


Figure 25: Scheme of the workspace adaptation for the KUKA LWR 4+

After this workspace selection, the available number of viewpoints is drastically reduced. An option to obtain more reachable viewpoints is to change the viewpoint sphere radius for each pose so it can fit inside the available workspace (Figure 26). This method deforms the final shape of the viewing poses, but their projections to the original view sphere remain unchanged, so the points are uniformly sampled.

At this point, all the output data of the *sphere_sampling* service has been computed, so it creates an array of *kinfu_msgs/KinfuPose* messages with the adapted viewpoints for the

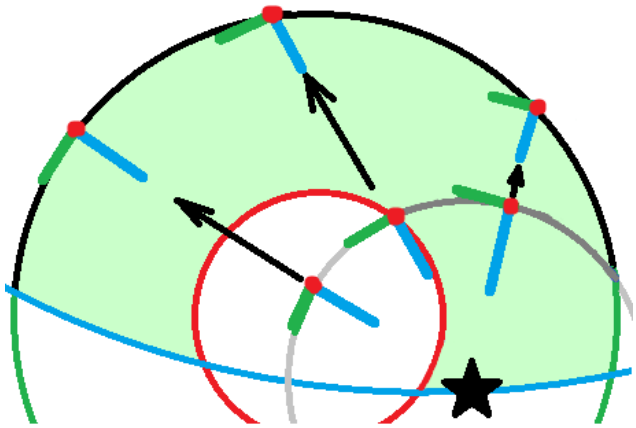


Figure 26: Scheme of the viewpoint radius adaptation

workspace. Note that the order of this viewpoints inside the array will be important in the next steps, because the viewpoints are always referred by their index.

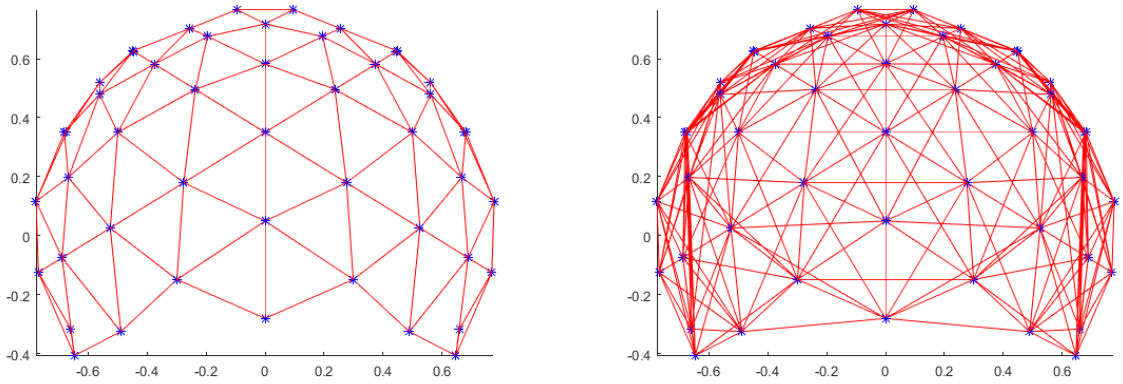
3.4.5 Graph Building

Once the viewpoints are generated, the same function that created them builds a graph that connects them and saves it. This graph is necessary for the *dijkstra* service which will be later executed.

The graph is created connecting the neighbour viewpoints and it is stored as an array of unidirectional edges, defined by the two viewpoint indexes that this edge connects. To make a bidirectional edge, two edges need to be created.

The neighbour connections can be tuned by two parameters: *n_neighbours*, that sets the maximum number of neighbours of each viewpoint, and *neighbour_tolerance*, that modifies the necessary relative distance to consider a near point as a neighbour. The larger this numbers are set, the more interconnected the graph will be (Figure 27). In our case, a very interconnected graph is preferred in order to move faster if the nearest points are not interesting, so this parameters are set as $n\_neighbours = 10$ and $neighbour\_tolerance = 3$.

As it is explained in Section 3.4.3, some movements are not desired because of joint limits and orientation issues. For this reason, some of this graph connections need to be erased: The sphere back area is disconnected, so to cross it the robot needs to go around the sphere



(a) Graph with $n\_neighbours = 6$ and $neighbour\_tolerance = 1.2$

(b) Graph with $n\_neighbours = 10$ and $neighbour\_tolerance = 3$

Figure 27: Graphs of the adapted viewpoint sphere, view from above

in the opposite direction. Moreover, to access the points to this back area, the robot needs to always cross the upper point in order to maintain smooth transitions. The resulting graph is shown in Figure 28.

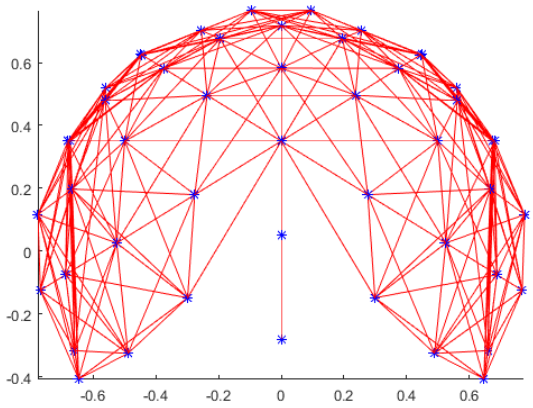


Figure 28: Limited graph taking the joint limits into account

3.4.6 Best-Path Planner

The last step where MATLAB actuates is computing the best path using a Dijkstra algorithm [32]. The service *dijkstra* is fired by the *planner_holder* class, and it asks for the

best path to follow towards the next best view.

Different variations of the Best Path planning are taken into account depending on the chosen objectives at every step (Section 2.4.3). However, this MATLAB layer only performs a Dijkstra algorithm so the *planner_ holder* is in charge of these variations.

In this Best Ppath planner step the previously saved viewpoint graph is imported. This service launch a Dijkstra algorithm using this graph and computing a cost function based on the information from the ROS node: the number of known and unknown voxels for each viewpoint. This cost function can be modified and the distance between viewpoints in the graph can also be taken into account.

The first approach to compute the cost function was to use the most relevant data: the number of unknown voxels ($vox_{unknown}$) that determines the best views. However, as this data is wanted to be maximized, Dijkstra is not the most appropriate algorithm because it uses a heuristic minimization procedure. Even if this cost is given as negative, the cumulative nature of this heuristic does not always find the optimal path. Moreover, this behaviour is not desired: The objective is to reach the target crossing the best points in the path and not to pass through all points until reaching the objective. For this reason, the number of known voxels (vox_{known}) is also used in the equation. The cost function, then, is computed by the normalized difference between the number of known voxels and the unknown ones:

$$cost = \frac{(vox_{known} - vox_{unknown})}{vox_{total}}$$

where vox_{total} is equal to the sensor total number of pixels and if all the rays are pointing towards the reconstruction scene, $vox_{total} = vox_{known} + vox_{unknown}$.

With this cost function the path will go more directly towards the objective when the views are good or it will try to pass through certain viewpoints if they are pretty bad. As the cost is mostly positive (usually $vox_{known} > vox_{unknown}$), there is no reason to add a distance related term. On the other hand, the cost can be eventually negative, that is not a problem: The algorithm will prefer going through this viewpoint but the focus will be kept towards the final objective.

Finally, a small change has been introduced to stabilize the behaviour when Dijkstra is executed iteratively at every point in the path. When doing that sometimes the best path was to go again towards the last visited viewpoint so a looping behaviour could be reached for some iterations. To avoid that, the service also provides the last visited viewpoint and the cost to go again from the actual position is increased.

3.5 Other Packages Modifications

Apart from the *planner* and MATLAB nodes, some other packages were needed to be adapted to work together and tuned to improve the system's performance. In this section this modifications are commented.

3.5.1 *kinfu* node

The *kinfu* node allows to perform a lot of different utilities. Some of them can be achieved in the runtime through commands or requests, but there are also some features that need to be hard-coded in a parameters file or specified in the launch file when starting the node. In order to keep the original code from the Github repository, this parameters have been written in the launch files:

- *forced_tf_current_frame = "kinect1_link_frame"* or *"kinect2_link_frame"*: This parameter allows to disable the internal tracking (ICP) and use an external TF instead. This parameter gives the name of the link to track, so it is different depending on which Kinect is used for the reconstruction. However, the ICP tracking is not disabled just with this parameter: the *COMMAND_TYPE_SET_FORCED_TF_FRAMES* command needs to be set to *True*.

- *forced_tf_reference_frame = "world"*: This parameter is also needed for disabling the ICP tracking. It sets the frame in which the *forced_tf_current_frame* is referenced.

- *shift_distance = 2.0*: This parameter tunes the behaviour of the *Kinect Fusion* algorithm related to shifting the TSDF volume when it is performing the reconstruction

(Section 2.3.1). This shifting procedure is useful when scanning large scale models such as room interiors, but in this object reconstruction task this behaviour is not desired. For this reason, increasing the *shift_distance* parameter the algorithm will not shift volumes if it is always pointing into the same volume.

3.5.2 *MoveIt!* KUKA model

To perform this task the model of the KUKA robot needed to be slightly modified as well as its environment model.

First of all, the Kinect holder needed to be modelled. To do it, a 3D mesh of a Kinect v2 has been downloaded and placed in the robot hand. However, as the holder contains both Kinect v1 and v2, this 3D model has been duplicated and placed in the correct position. For the precise location of this links, an eye-hand calibration needed to be performed (Section 4.1.1). To place the holder model, several fixed joints and links were defined as well as their collisions. The most important links are:

- *kinect1_link_frame*: The calibrated sensor position for the Kinect v1 to perform the forced TF option.
- *kinect2_link_frame*: The calibrated sensor position for the Kinect v2 to perform the forced TF option.
- *kinect_ee*: Middle position between both Kinects that is defined as the end effector for the KUKA robot, so it will place it in the asked viewpoints.

After that, the environment for the robot has been defined: It is located in a table and half sphere is placed in the position of the scanned object (Figure 29). By this way, collisions of the robot against the table or the scanned object are avoided.

Finally, the joint limits of the KUKA LWR 4+ have been adjusted for this task. This modifications are done to prevent the robot reaching the viewpoints with unnatural joint positions. As the orientations for the viewpoints have been previously studied (sections 3.4.3 and 3.4.4), the robot can reach them with easy and natural configurations. However,

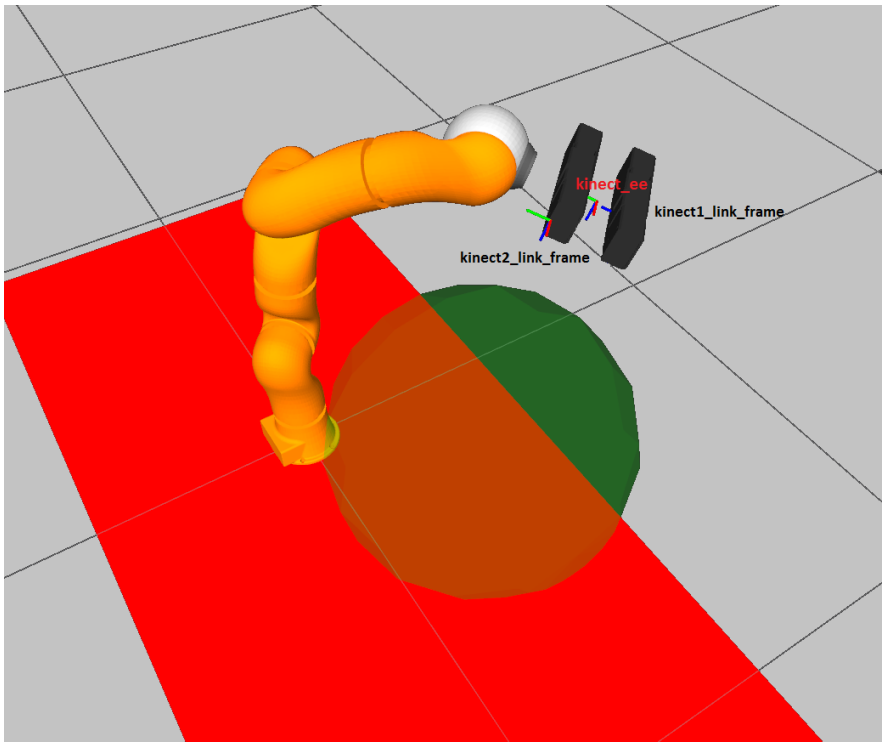


Figure 29: *MoveIt!* model of the environment and the Kinect holder, visualized in Rviz

due to joint redundancies and wide limits, KUKA might reach the poses with unnatural configurations near to the joint limits. That caused the robot to sometimes change completely its configuration when moving through near points, wasting a lot of time and losing the scanned object in the camera's field of view, which is terrible for ICP tracking. For this reason, the joint limits have been adjusted for this task, also fixing the redundant joint which is not necessary in this case.

4 Experimental Results

After all the system implementation has been explained, this section presents the real life set-ups and experiments, evaluating the different algorithm performances and comparing them.

4.1 Hardware and software set-up

The experiments have been performed in the Intelligent Robotics Laboratory, in TUAS building of Aalto University, Finland. This laboratory has a KUKA LWR 4+ mounted in a table, and a powerful computer with *CUDA* compatible graphics card so it can run *Kinect Fusion* algorithms.

The reconstruction scenario can be seen in Figure 30. It has the following elements:

- KUKA robot as the positioning system, mounted above a fixed table.
- Kinect holder mounted as the hand of the robot. This piece holds both Kinect v1 and Kinect v2, so both performances can be tested.
- The target object for the reconstruction, which simply lies above the table so no extra fixtures are needed.
- Chess pattern sheets fastened to the table in a known position. They are used for both intrinsic camera and eye-hand calibration (Section 4.1.1).
- Auxiliary objects in the table, that are useful for helping the ICP tracking algorithm adding extra features in the scene. By this way, it is more difficult for the tracking to lose references.

Figure 31 shows the software interface in *Rviz* for the same scenario as Figure 30. On the bottom left window the *kinfu* reconstructed 3D is shown at real-time. On the right window, the *MoveIt!* model of the robot, the holder and the environment are shown. The computed

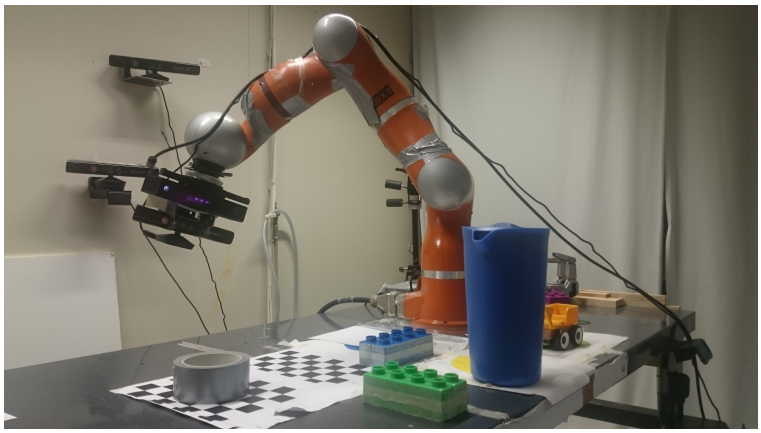


Figure 30: Aalto University Intelligent Robotics Lab

viewpoints from MATLAB are also shown, as well as the RGB pointcloud of the Kinect published by the *openni* node (Kinect v1) or *kinect2_bridge* node (Kinect v2).

There are two markers in the visualization: a sphere and a bounding box. The sphere represents the minimal distance to the object and contains a non-visible sphere to avoid the robot collisions with the object. The bounding box crops the reconstructed 3D model, so only the model inside it is saved at the end of the task. All these parameters are easily configurable in the *rqt_reconfigure* window.

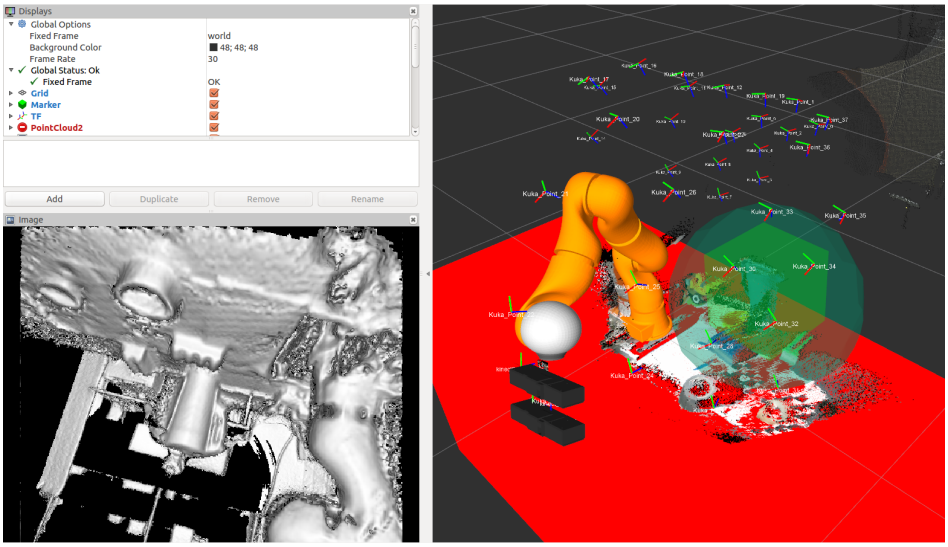


Figure 31: *Rviz* software interface

4.1.1 Kinect Holder and calibrations

To mount both Kinects in the KUKA's hand a holder for them has been adapted (Figure 32). This holder is a metal piece that strongly fixes Kinect v2 inside. An aluminium plate has been attached to the bottom of the holder in order to fix it to the KUKA's fixture. Finally, Kinect v1 has been attached upside-down at the top of the holder.

Figure 32: Kinect holder mounted in the KUKA's hand

Once the hardware is mounted, the cameras need calibration in order to achieve precise results [8]. Two types of calibrations have been done to the sensors: Camera (or intrinsic) calibration and eye-hand (or extrinsic) calibration.

4.1.1.1 Intrinsic camera calibration

The intrinsic camera calibration corrects the distortion of the camera due to the lens. By this way, a good calibration will lead to a more accurate reconstructed model. As Kinect sensors have two different cameras (a RGB normal camera and an infrared one) both cameras can be calibrated. However, as the *Kinect Fusion* algorithm only works with depth information, only the infrared camera needs to be calibrated for the reconstruction task. Figure 33 shows the effects due to lens distortion of a non-calibrated Kinect v2 sensor.

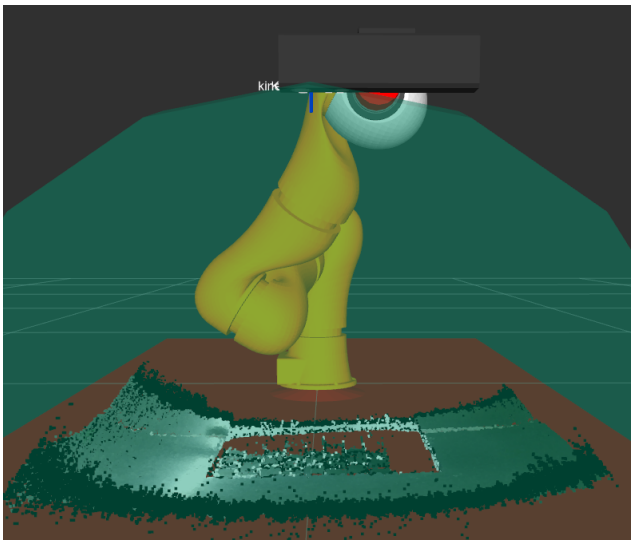


Figure 33: Pointcloud of a not calibrated Kinect v2

To calibrate the IR camera the chess patterns glued in the table are used. Using a special program for the calibration, the camera was placed in different positions and orientations pointing towards the chess pattern (Figure 34). The program detects the pattern and applies corrections if the known distance between squares is not the correct one. After that, this distortion parameters are saved and will be loaded when the camera program starts to correct the input.

4.1.1.2 Extrinsic eye-hand calibration

Once the camera is intrinsically calibrated, it is important to locate the sensor reference in relation to the robot arm. This type of calibration is known as eye-hand calibration, and can be also performed with the same chess pattern used before. This calibration is very important if the ICP tracking is disabled, because the camera reference is used for the reconstruction and thus, it has to be very accurate. However, when ICP is active only an approximation is required to initialize the algorithm in a certain position.

As in this work the ICP is mostly used, eye-hand calibration has been done manually (Figure 35). If more precision was required, an automatic program similar to the one used in the intrinsic calibration would be executed at the beginning of the task.

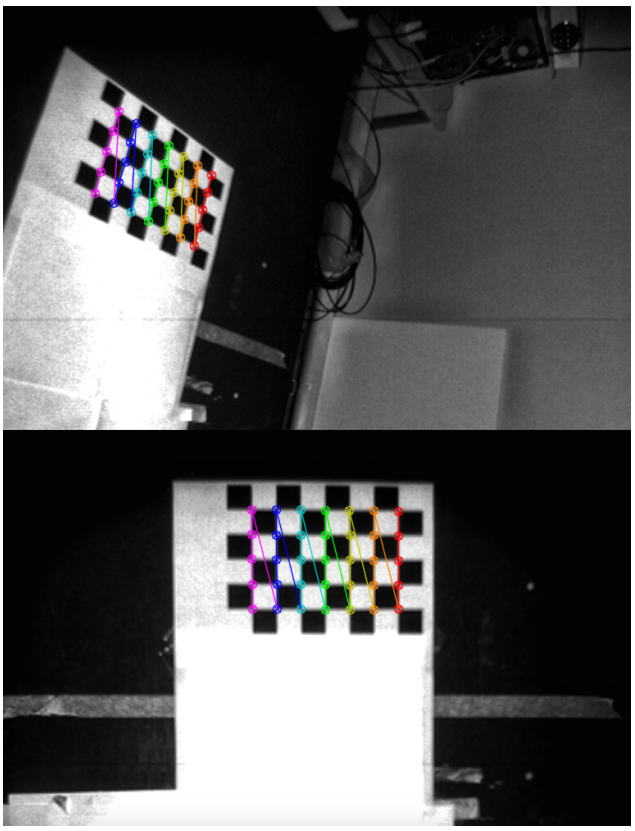


Figure 34: Output of the calibration program, detecting the chess pattern in different positions and orientations

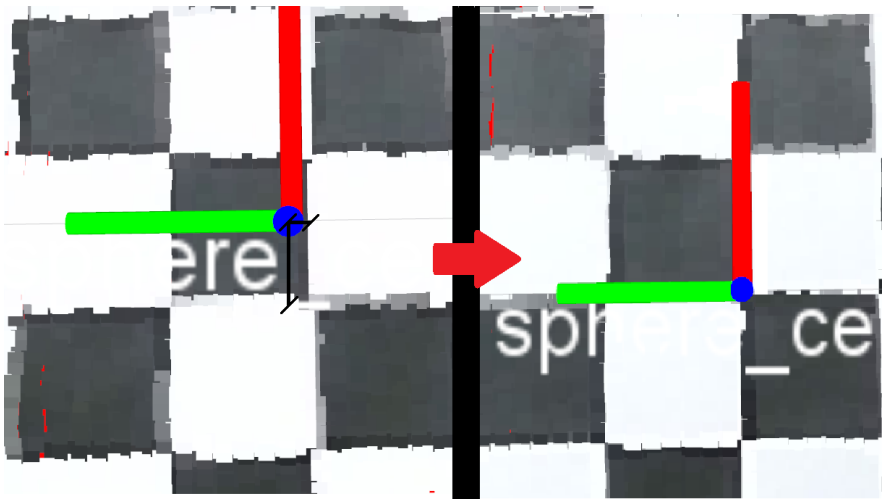
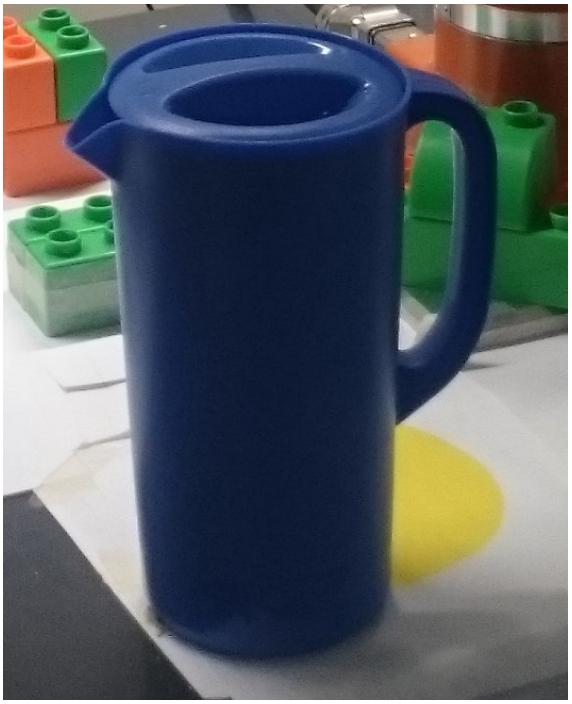


Figure 35: Using a known reference, the Kinect link position is visually calibrated

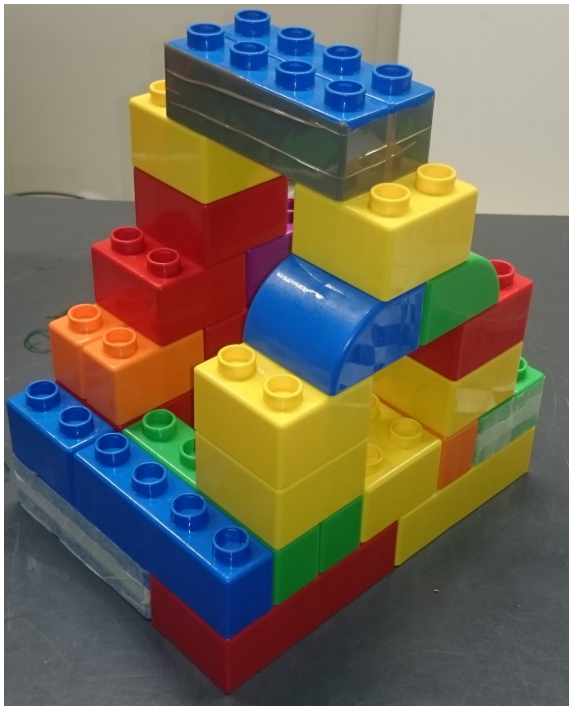
4.1.2 Scanned Objects

In order to evaluate the performance of this algorithms, four different objects have been chosen to reconstruct. All of these have some special properties that make them interesting to test:

- A pitcher (Figure 36a): This model included in the YCB model set [33], so the reconstruction can be compared to a ground-truth model. Its handle can be hard to model, so special attention will be taken there.
- A complex LEGO building (Figure 36b): Thanks to the LEGO blocks versatility, a complex model with different internal concavities has been built.
- A mug (Figure 36c): Due to its small size, Kinect's low resolution can make it difficult to model. It also has a handle and its internal concavity is interesting to check the reconstruction quality.
- A ball with occlusions (Figure 36d): A ball is quite a simple object, but in this case it is partially occluded so is interesting to check which areas are able to be modelled.

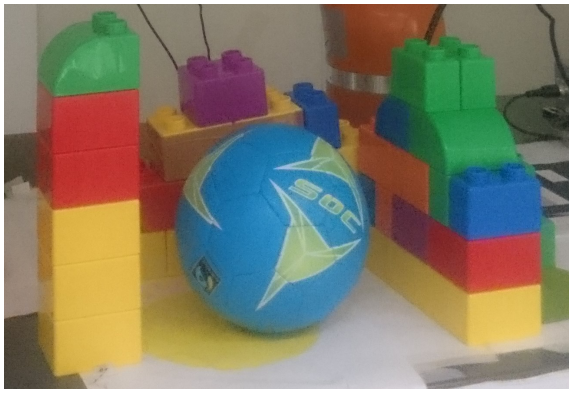


(a) Pitcher



(b) Complex LEGO building

(c) Mug



(d) Occluded ball

Figure 36: Scanned objects in this project

4.2 Experimental Configuration

Before doing the set of tests with all the objects, some previous tests have been performed to check the reconstruction quality with the different configurations.

4.2.1 Kinect v1 against Kinect v2

As it is explained in Section 2.2.1.1, Kinect v1 and v2 perform different technologies. For this reason, the two sensors are compared to check which performs better with the reconstruction tasks.

It is known that active triangulation based sensors are widely used in 3D reconstruction. For this reason, Kinect v1 seems a good option for this task. However, Kinect v2 has much better specifications such as more resolution or greater field of view, so it is also tested.

When first evaluating the static 3D model of both Kinects, the v2 builds a much more defined world model than v1. However, when they move, the sensed world of Kinect v2 is deformed due to the perspective changes, so the object locations are shifted some centimetres respect to the already stored 3D model. This causes the redefinition of the model, so the object is erased on one side an built again at the other. This effect makes the reconstruction to perform deformed results (Figure 37), so additional image processing should be applied to counteract this effect.

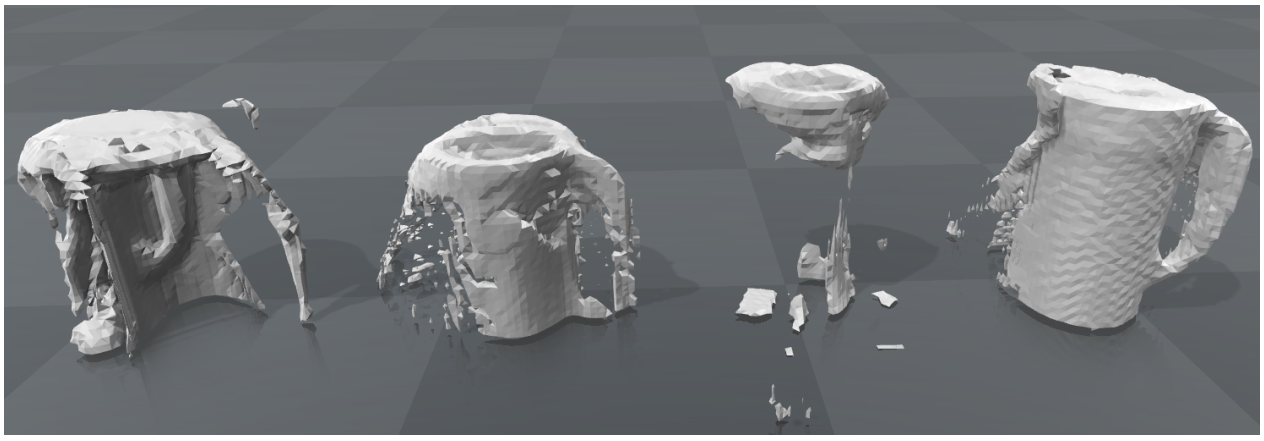


Figure 37: Four different trials to scan the pitcher with the Kinect v2

On the other hand, Kinect v1 performs correctly when moving. As the objective of this project is to evaluate the view planning algorithms and obtaining an entire model regardless of its resolution, Kinect v1 is chosen to perform the next tests.

4.2.2 Forced TF against ICP

Once Kinect v1 is chosen for testing, the performance between forced TF and ICP tracking is evaluated.

Iterative closest point (ICP) tracking is an effective method to track if there is no information where the camera is. However, it has some drawbacks: It can lose the references due to high speed or too few features in the model so it would stop working. In addition, it can confuse the features of similar points and make the reconstruction model invalid.

On the other hand, if a known reference frame of the camera is fixed (Forced TF) all the ICP problems are solved. Nevertheless, this method absolutely relies on the precision of the reference, so an imprecise one would completely worsen the 3D model.

After several tests with both methods (Figure 38) we can conclude that the eye-hand calibration is not perfectly accurate and the holder sometimes can slightly change its rotation due to the cable tension, so the Forced TF method does not give precise results in this implementation. For this reason, all the tests will be performed using the ICP rather than the forced TF. Moreover, if this system is implemented into a mobile robot [23], an accurate camera reference would be difficult to get so ICP would be the most appropriate option.

If the camera reference is not very precise, the discrete reconstruction option (stop scanning when the camera moves) is neither an appropriate option to choose, so the tests will be done in continuous reconstruction. In addition, continuous reconstruction is reported to perform better results than discrete for *Kinect Fusion* in R. Monica's work [16].

4.3 Next-Best-View Approach

In the first place, the Next-Best-View approach implemented in R. Monica's work [16] is evaluated for the four objects.

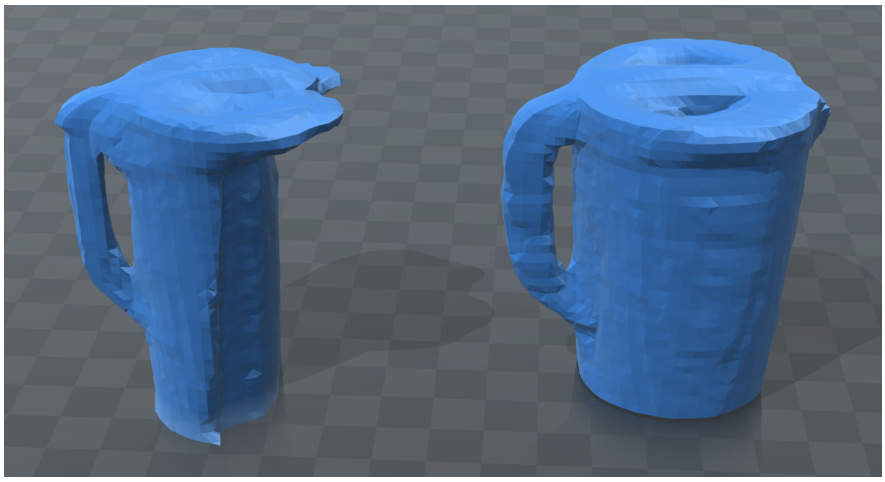


Figure 38: Two pitcher models scanned with Kinect v1, using forced TF (left) and ICP (right)

Figure 39 shows the reconstructed model of the pitcher using NBV starting from different positions. As it can be seen, the reconstruction quality is high except in the last model. While building this last model the ICP tracking was lost for some moments, so the result is erroneous. This is the principal problem for the NBV planning: As only the final points are given, and they are usually far from the actual position, the *MoveIt!* motion planner moves the robot in a non-optimal way to keep tracking of the scene. This usually makes the algorithm to lose the references, and once this happens is difficult to obtain a precise reconstruction. This reference-losing behaviour often happens when the camera turns over (pointing to the rest of the room and the ceiling) or when it moves closer to the object than the recommended minimum range.

Figure 40 shows two different views for the reconstructed model of the complex Lego structure. This model presents an accurate view of the model from one side, but the other shows small deformations. The problem again is the difficulty of the ICP algorithm to track the NBV movements. However, the final result is accurate: it shows the correct shape of the object as well as the internal concavities. This happens because this Lego model has a considerable size and Kinect v1 is able to scan it well. On the contrary, when performing the reconstruction of a small object like the mug (Figure 41) the low resolution of the sensor builds a model losing a lot of properties. In the mug model the handle is too small to be

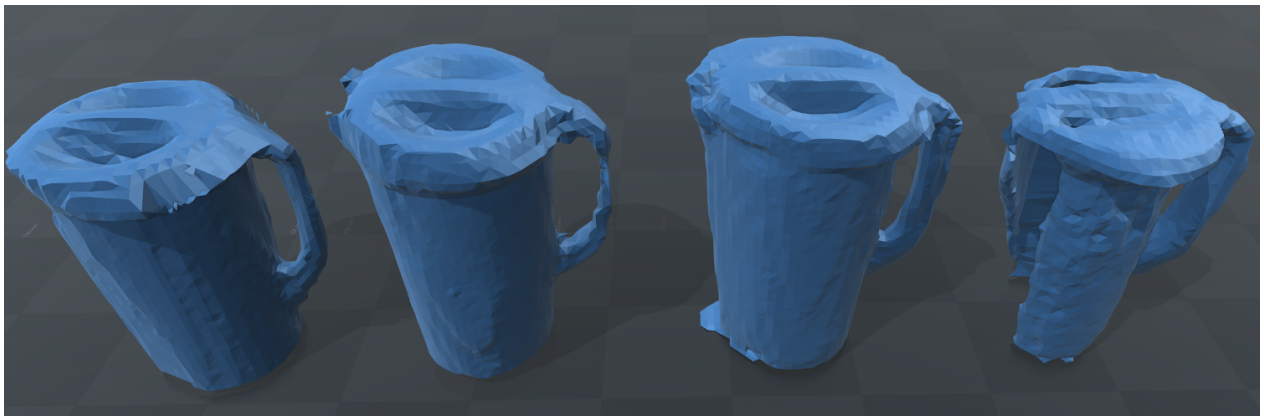


Figure 39: Four different pitcher models scanned using NBV algorithm starting from different positions

detected, and the NBV movements makes the ICP tracking loosing other properties so the convex shape is partially destroyed.

Regarding to the NBV movements, two paths are shown in Figure 42. Even if sometimes this behaviour is not followed, this algorithm usually goes to new viewpoints close to already scanned ones in the past. This diminished the algorithm performance as the robot needs to go back and forth to the same areas. For this reason the designed method in this project, explained in Section 2.4.3, tries to avoid this attitude optimizing the scanning time and the model quality.

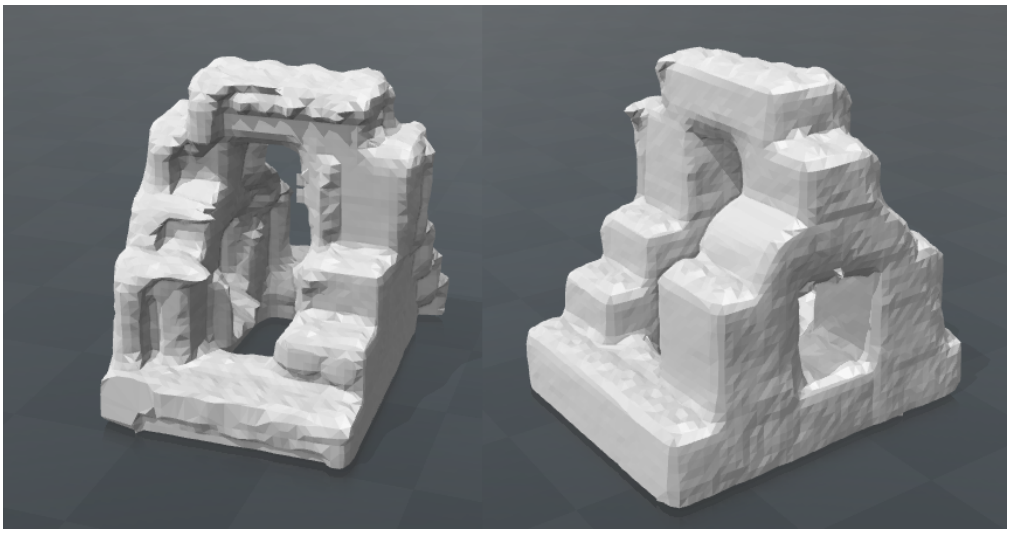


Figure 40: Two views of the complex Lego model scanned using NBV algorithm

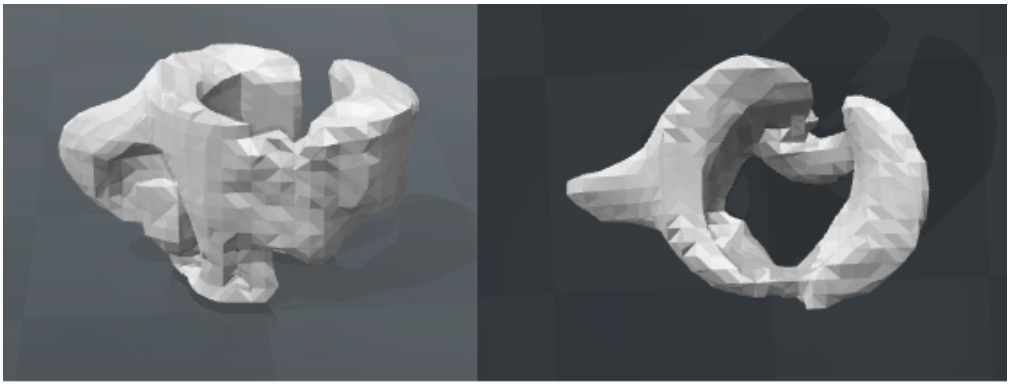


Figure 41: Two views of the small mug model scanned using NBV algorithm

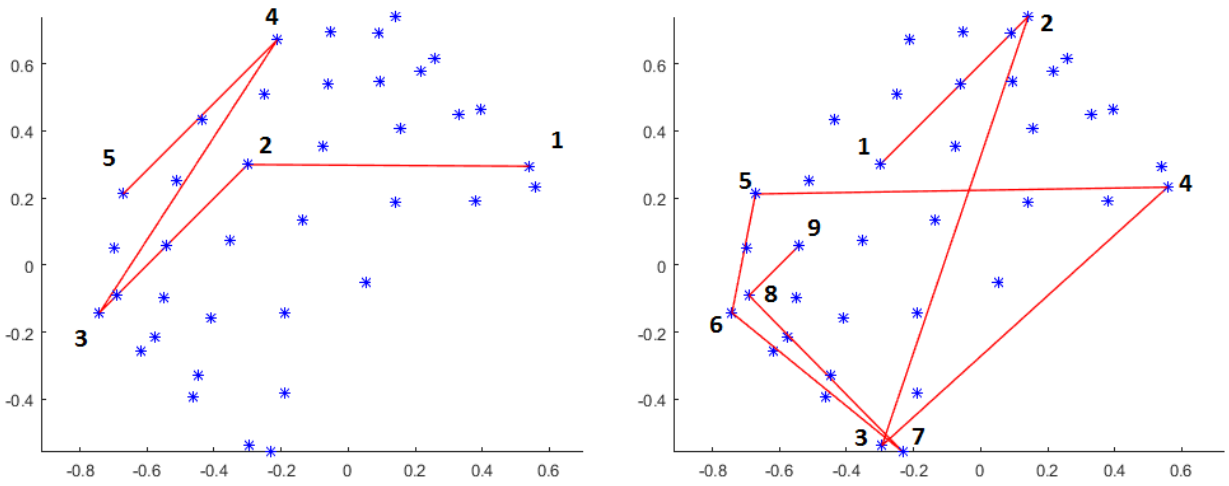


Figure 42: Followed scanning path using NBV in the pitcher object (left) and on the complex Lego structure (right)

4.4 Best-Path Approach

After seeing the NBV disadvantages, the BP Algorithm implemented in this project is tested and evaluated. As explained in Section 2.4.3, this approach uses the Dijkstra algorithm to go to the next best view going through the best viewpoints in the path.

Implementing the Dijkstra algorithm for this task the three defined behaviours in Section 2.4.3 were tested:

- *Dynamic Dijkstra*: It goes just to the next viewpoint in the path. Once there the next best view is computed again and a new Dijkstra is computed pointing toward this new objective.

- *Dijkstra NBV*: It computes the Dijkstra path towards the next best view, and follows this path until reaching the end so it is not dynamically recomputed.

- *Dijkstra Dynamic NBV*: It computes the Dijkstra path towards the next best view, that is saved until this objective is reached. At each viewpoint in the path, a new Dijkstra path is computed pointing to the previous saved objective.

This three behaviours have been tested for all four objects, with several repetitions each one. All this approaches using the BP planning presents similar output models because the viewpoints are the same. However, they present slight differences regarding the operation time, which are commented in the discussion Section 4.5. The following figures show the best model for each of the four tested objects, obtained using this BP algorithms.

Figure 43 shows the pitcher model. This model is quite accurate because it has a continuous and smooth surface with no holes. The handle have also been correctly modelled as well as other details.

Figure 44 shows the complex Lego model. Unlike the previous NBV scan for the same object, now the surface is smoother while the concavities have been also successfully modelled.

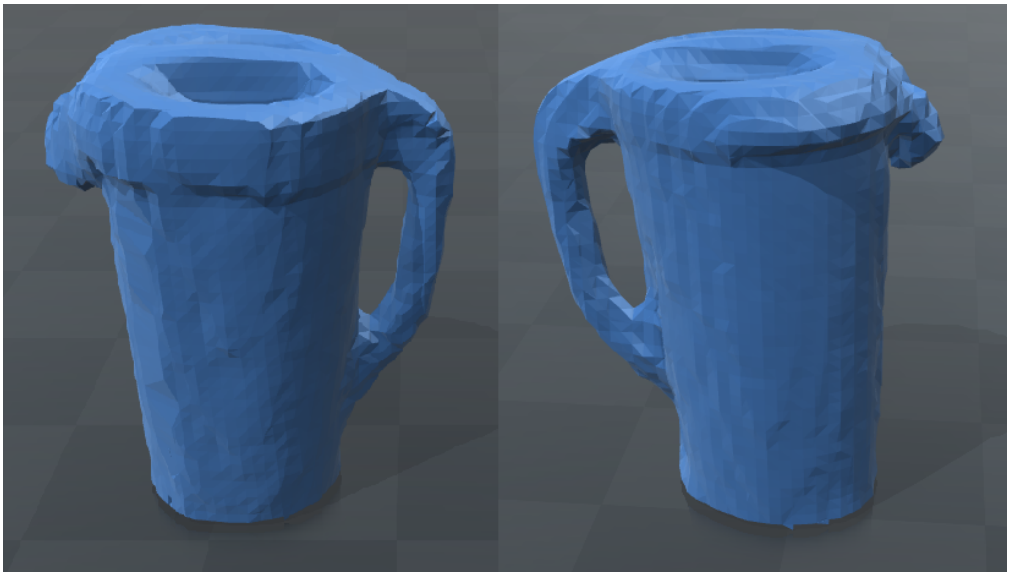


Figure 43: Pitcher model scanned with BP algorithms

Figure 44: Complex Lego model scanned with BP algorithms

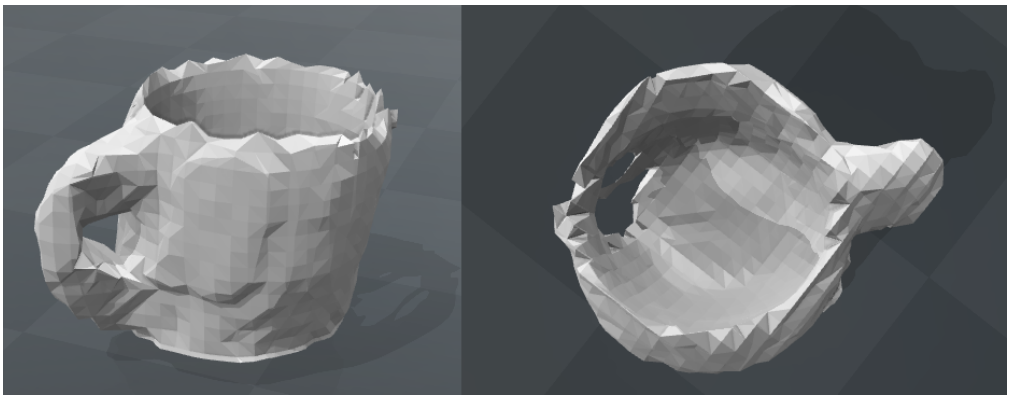


Figure 45: Small mug model scanned with BP algorithms

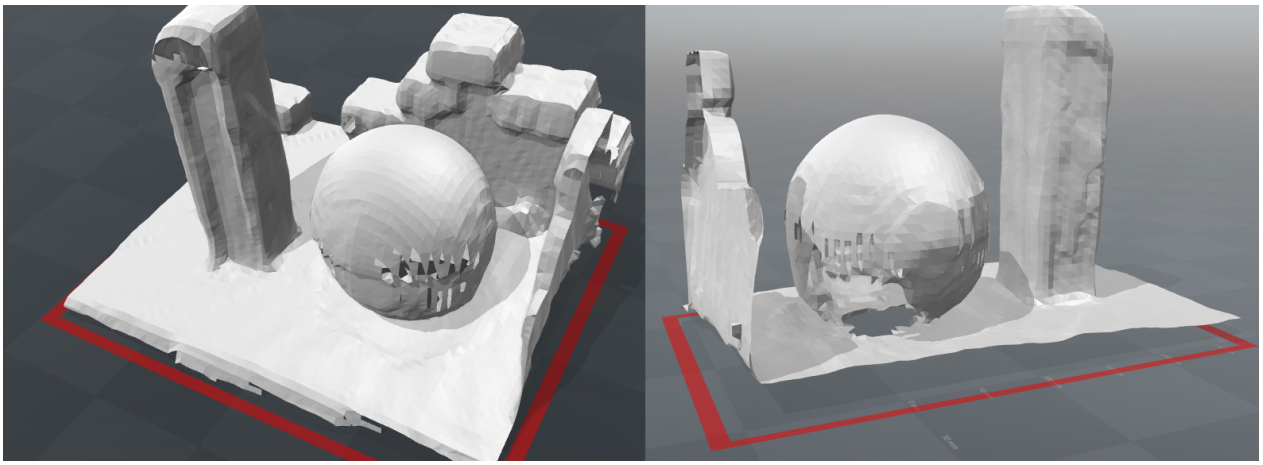


Figure 46: Hidden ball model scanned with BP algorithms

Figure 45 shows the small mug model. In this case the model is much more accurate: The shape is nearly continuous (but still not perfect) and the handle hole have been detected.

Figure 46 shows the hidden ball model. The scanning have been successful for this object so most of complicated hidden areas have been modelled. However, it is logical that some of the most hidden areas still are unknown due to occlusion and high incidence angles that are difficult to detect.

Finally, Figure 47 shows the scanned path for the pitcher and Lego models. The path has a lot of middle points, but it can be seen how the the movement follows a general direction that covers all the views.

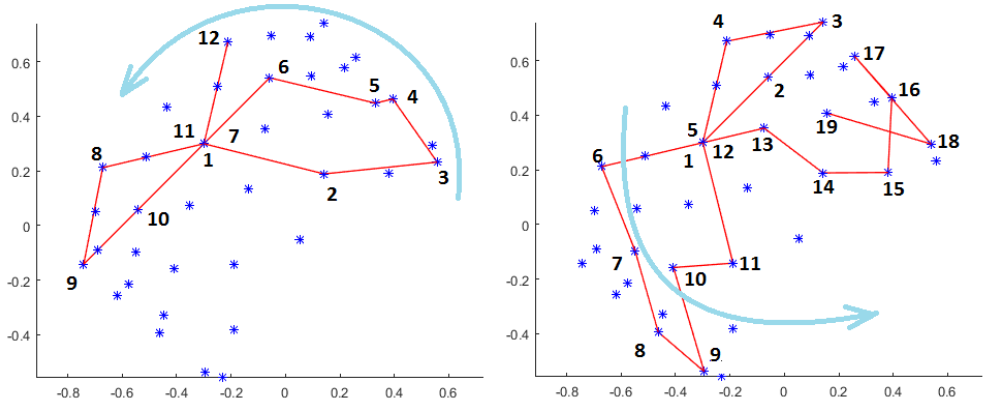


Figure 47: Followed scanning path using BP in the pitcher object (left) and on the complex Lego structure (right)

4.5 Discussion

Once the NBV and BP algorithms models have been shown, in this sectionthe performance of both is discussed. Moreover, the three different BP behaviours are compared, choosing the best one for each situation.

Three main aspects for each algorithm are evaluated:

- Scanning robustness: The most important characteristic is if the model is complete. If the model presents big discontinuities or deformed surfaces it is not taken into account. This aspect considers the ratio of successful scans versus the number of attempts.

- Model precision: If the deformations of the model are small the model is accepted. Firstly a visual inspection is performed to compare the model quality. In the pitcher object its precision is also compared towards its ground-truth model with a special software.

- Scanning time: The scanning time is compared so the fastest method will be found. However, this aspect is not very relevant in some cases because deformed models are usually scanned faster.

- Repeatability: Finally, the variation in the measured times is evaluated: Low variation

means that the algorithm has high repeatability, so the scanning times will be consistent over different trials.

Next-Best-View vs Best-Path

Firstly the Next-Best-View (NBV) implemented by R. Monica [16] is compared towards the Best-Path (BP) planning developed in this project. This new approach presents a lot of advantages when it is used among the ICP tracking:

- BP Planner is more robust: Only 60% of the scanning trials ends in a good model when using NBV. This is because of the easy ICP track loosing when the algorithm is not taking the camera movements into consideration. On the other hand, BP planning moves the camera between near viewpoints so it is always pointing the scanned object. This causes the BP successful scanning rate to stay higher than 90%.

- BP gives more accurate models than NBV planning. This is due to the same ICP performance problems: Even if the visual tracking is not lost, small shifting can cause the model to loose precision. This effect can be seen comparing Lego object surfaces (Figure 40 and Figure 44) or checking the overall performance scanning the mug (Figure 41 and Figure 45). To compare this precision in absolute terms, the pitcher model of both NBV and BP is compared towards a ground-truth model from the YCB model set (Figure 48). The mean distance from the registered points is -1.762 millimetres for the BP model and -2.068 millimetres for the NBV one.

- BP is more time-efficient, because the paths are optimized and there are less chances to return to neighbour viewpoints in the future. Figure 47 shows how the path is following a circular tendency exploring the areas in an organized way. However, Table 2 shows the opposite results: It says that NBV is faster in some objects. This happens for two main reasons:

  - Due to the movement velocity: all the experiments have been performed with the same movement speed, so BP scans slower because it stops to more intermediate

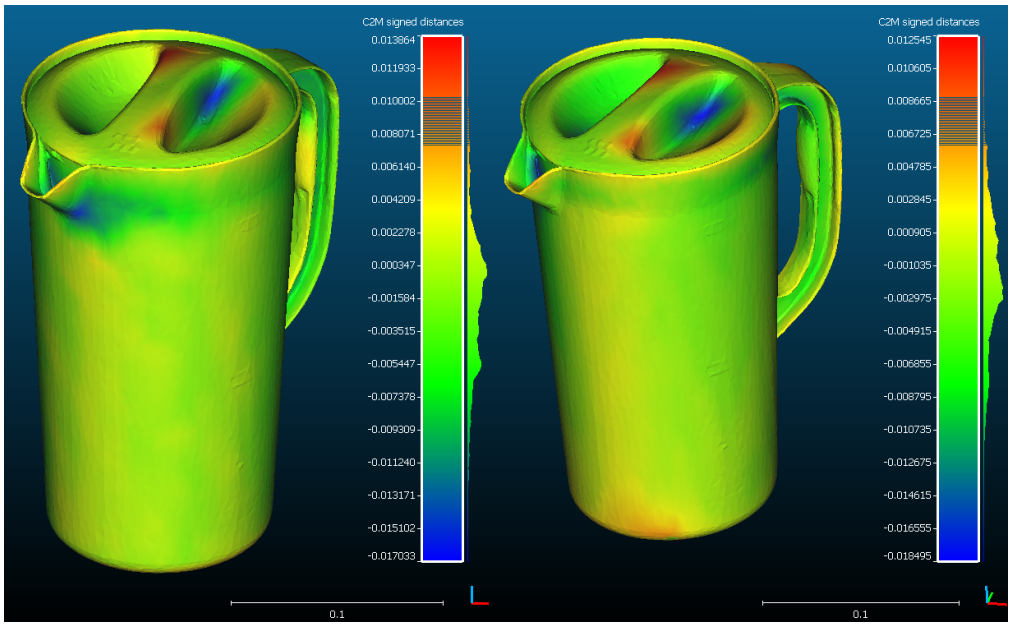


Figure 48: BP distance error (Left) and NBV distance error (Right) with respect to the ground-truth YCB model [33] using the open source software *Cloudcompare* [34]

viewpoints and there is a delay due to its non-continuous nature. This can be solved interpolating the movements to follow a continuous trajectory. Moreover, its movement speed can be increased because it is always pointing towards the object so its difficult to lose the ICP tracking.

- For getting worse models: When NBV have some scans with deformed results, the shifted voxels that gives the wrong information are considered as known, so the scan finishes earlier but giving a worst model as an output. If the model is accurate, there will be some voxels that are difficult to see so the algorithm will need more points of view.

- BP has higher repeatability: Looking in Table 2 it can be seen how the 95% confidence interval values for the NBV algorithm are higher than the BP ones. All of them are over 30% with respect to the mean, so the experiment times of NBV vary from very low values (when it is obtaining a worse model) to higher times (when the model is accurate, but the algorithm moves in an inefficient way).

Object	NBV	BP		
		Dijkstra NBV	Dijkstra Dynamic NBV	Dijkstra Dynamic
Pitcher	$92 \pm 34\%$	$103 \pm 25\%$	$134 \pm 21\%$	$129 \pm 24\%$
Lego	$208 \pm 32\%$	$235 \pm 21\%$	$252 \pm 26\%$	$224 \pm 23\%$
Mug	$61 \pm 36\%$	$49 \pm 31\%$	$70 \pm 32\%$	$58 \pm 29\%$
Ball	$113 \pm 31\%$	$142 \pm 24\%$	$130 \pm 27\%$	$119 \pm 20\%$

Table 2: Mean time and 95% confidence interval of the experiment times, in seconds

Best BP behaviour

Several tests have been performed with all the objects and performing the three different BP behaviours. Table 2 shows the statistics of the experiments to compare which is more time efficient. Scanning robustness and model quality are not compared between this behaviours, because they perform very similar results in this fields as they are based in the same BP algorithm.

All three behaviours have similar results that slightly change over the different objects. In the first place, *Dijkstra NBV* obtains accurate results with the pitcher and mug objects. This is due to their simpler surfaces: If the surface is simple, it will be recognized from more viewpoints, so it is not necessary to explore local maximums because they will be modelled from the direct path to the next best view. On the other hand, Dynamic algorithms work better in complex environments such as the Lego structure or the hidden ball object, where exploring local maximums is more optimal because this specific surfaces will not be seen from anywhere else.

Comparing *Dijkstra Dynamic* against *Dijkstra Dynamic NBV* it can be seen how the first one is always faster. This is a logical result, as *Dynamic NBV* is exploring local maximums areas but still going to the old global best view, that in some cases can be obsolete and going there is a waste of time. *Dijkstra Dynamic* avoids that, because the next best view objective is recomputed at every iteration.

5 Conclusions

The objectives of this thesis were multiple: Firstly, the *Kinect Fusion* software needed to be implemented in our custom built environment as well as the NBV algorithm. After that, an original view planning method was required to be added to try outperforming the previous one. Finally, the system needed to be implemented and tested in a real robot and compare the results. All these goals have been successfully achieved by the end of this project.

5.1 Summary

Initially, the state of the art have been explored to find an economical, robust and easy to implement solution to build a complete reconstruction system: The *Kinect Fusion* algorithms (see Section 2.3). After deeper research, the RMonica version of the original *Kinect Fusion* was found, which includes interesting improvements with respect to the original algorithms (see Section 2.3.2).

After deciding to use the RMonica *kinfu* version, a complete suite was built to easily operate with this ROS node. This program included a *rqt_reconfigure* window to easily operate the *kinfu* nodes. Next step was implementing the NBV algorithm and creating a new optimized view planning algorithm: the Best-Path planner. This new planner used different variations of the Dijkstra algorithm to outperform the NBV performance.

The original suite to command the *kinfu* node was gradually updated until reaching the final implementation shown in this project (see Section 3). This complete system explores the different capabilities of ROS *dynamic reconfigure* servers as well as the MATLAB-ROS integration. MATLAB was used to operate with the viewpoints positions and orientations, which were studied in detail in order to optimize the positioning system and allowing the robot to move in a natural way. The final system is also connected to *MoveIt!* motion planner, which adds the possibility of working with the KUKA LWR 4+ robot.

After that, the system was ready to execute the real 3D reconstruction tasks so a comparison of the different view planning algorithms was performed (see Section 4.5). Several

improvements of the new Best-Path planner were proved with respect to the NBV:

- BP is more robust, because most of the scans end in a valid model whereas NBV does not, due to ICP tracking conditions.
- BP gives more accurate models, due to the same ICP tracking problems that NBV carries.
- BP is more time-efficient, because the paths are optimized and there are less chances to return to neighbour viewpoints in the future.
- BP has higher repeatability, because all the scanning times for the same object have less variability than the ones with NBV.

In addition, three different variations of the Best-Path planner were tested in order to find the most optimal one (see Section 2.4.3). Looking to the scanning times of all these variations, we can conclude that *Dijkstra NBV* is more optimal in simple objects, whereas *Dijkstra Dynamic* is better with complex ones. *Dijkstra Dynamic NBV* has been proved not to be worth it, because has a performance like *Dijkstra Dynamic* but always a bit worse.

5.2 Future Work

This thesis opens a lot of possibilities to work with the reconstruction system. After having an operative 3D scanning environment there appears several ways to improve the system or finding new utilities. In this last subsection, some ideas are given to improve the built programs and adding extra features.

In the first place, an automatic and precise eye-hand calibration would be useful to implement in the current system. By this way, if Kinect frames are very precise new tests could be performed using the *forced_tf* option, so most of the ICP problems would be solved.

Secondly, an improvement of the Best-Path planner can be achieved using polynomial interpolation between the movements so that the path can be converted into a smooth trajectory. This approach would decrease the scanning times and more natural movement

would be achieved. However, this variation would include some challenges regarding to dynamic algorithms that need to be computed every time the camera reaches a viewpoint.

Another feature to improve the current system would be to add extra processing to counteract the undesired shifting effects of the Kinect v2 camera. These effects destroyed the build models when the camera was moving, so in the end Kinect v2 has not been taken into account for this project. However, if this effects were solved the final model would have much greater definition and the scanning would be more accurate thanks to Kinect's v2 better specifications.

Finally, the future work where this thesis is pointing to is implementing this scanning system into a mobile robot for grasping purposes. To achieve that, firstly the system would need to be merged with a grasping system, thus the performance can be tested with the KUKA LWR 4+ robot. After that, the complete system would be ready to be implemented in a mobile robot with integrated eye-in-hand range cameras and grasping tools in its arms. A good testing platform can be the recently purchased Care-O-Bot 4, which opens a world of possibilities in home-assistance mobile robots.

References

[1] Aldoma, A.; Prankl, J.; Svejda, A.; Vincze, M.; "Object Modelling with a Handheld RGB-D Camera" *OAGM Workshop*, Automation and Control Institute, Vienna University of Technology, Austria, 2015.

[2] Aleotti, J.; Rizzini, D.L.; Monica, R. & Caselli, S.; "Global Registration of Mid-Range 3D Observations and Short Range Next Best Views", *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014)*, Chicago, IL, USA, September 14-18, 2014.

[3] Banta, J.E.; M.Wong, L.; Dumont, C. & Abidi M.A.; "A Next-Best-View System for Autonomous 3-D Object Reconstruction". *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans. Volume: 30, Issue: 5*, Sep 2000.

[4] Besl, P.; "Range image sensors. In Advances in Machine Vision", J. Sanz, Ed. Springer-Verlag, New York, NY, 1989.

[5] Chen, S.; Li, Y.; Kwok, N.M.; "Active vision in robotic systems: A survey of recent developments". *IJRR 30(11), 1343-1377* (2011).

[6] El-Hakim, S. & Beraldin, J.-A.; "On the integration of range and intensity data to improve vision-based three-dimensional measurements" In *Proceedings of the SPIE Conference on Videometrics III* (Boston, MA). Vol. 2350. 306–321, 1994.

[7] Fofi, D.; Sliwa, T.; Voisin, Y.; "A Comparative Survey on Invisible Structured Light". *SPIE Electronic Imaging — Machine Vision Applications in Industrial Inspection XII.* San Jose, USA. pp. 90–97, January 2004.

[8] Hartley, R. & Zisserman, A.; "Multiple View Geometry in Computer Vision". *Cambridge University Press. pp. 155–157. ISBN 0-521-54051-8*, 2003.

[9] Khalfaoui, S.; Seulin, R.; Fougerolle, Y.; Fofi, D.; "An efficient method for fully automatic 3D digitization of unknown objects". *Le2i Laboratory, UMR-CNRS 6306,* University of Burgundy, 71200 Le Creusot, France, Apr. 2013.

[10] Krainin, M.; Curless, B. & Fox, D.; "Autonomous Generation of Complete 3D Object Models Using Next Best View Manipulation Planning". In *Proceedings of the IEEE International Conference on Robotics & Automation (ICRA)* Shanghai, China, pp. 5031-5037, 2011.

[11] Kriegel, S.; Rink, S.; Bodenmüller, T.; Narr, A.; Suppa, M. and Hirzinger, G.; "Next-Best-Scan Planning for Autonomous 3D Modeling". In: *IEEE/RSJ IROS, pp.2850-2856.* Vilamoura, Portugal, 2012.

[12] Kriegel, S.; Rink, C.; Tim, B.; Narr, A.; Suppa, M.; & Hirzinger, G.; "Efficient next-best-scan planning for autonomous 3D surface reconstruction of unknown objects". *J.Real-Time Image Processing*, 1-21, 2013.

[13] LaValle, S.M.; "Rapidly-exploring random trees: A new tool for path planning". Computer Science Department, Iowa State University (TR 98-11), October 1998.

[14] McGreavy, C.; Kunze, L.; & Hawes, N.; "Next Best View Planning for Object Recognition in Mobile Robotics". University of Birmingham, United Kingdom, 2016.

[15] Mehlhorn, K.; Sanders, P.; "Chapter 10. Shortest Paths". *Algorithms and Data Structures: The Basic Toolbox. Springer,* 2008.

[16] Monica R.; Aleotti J. & Caselli S.; "A KinFu based approach for robot spatial attention and view planning", Dipartimento di Ingegneria dell'Informazione, University of Parma, Italy, 2016.

[17] Monica, R. & Aleotti, J.; "Contour-based next-best view planning from point cloud segmentation of unknown objects", Springer Science+Business Media New York, Jan. 2017.

[18] Newcombe, R.A.; Izadi, S.; Hilliges, O.; Molyneaux, D.; Kim, D.; Davison, A.J.; Kohli, P.; Shotton, J.; Hodges, S.; Fitzgibbon, A.; "KinectFusion: Real-Time Dense Surface Mapping and Tracking". In *Proceedings of the IEEE International Symposium on Mixed and Augmented Reality 2011 (ISMAR)*, Basel, Switzerland, pp. 127-136, 26-29 October 2011.

[19] Pito, R.; "A solution to the next best view problem for automated surface acquisition", *IEEE Trans. PAMI 21, 10 (Oct.), 1016–1030*, Oct. 1999.

[20] Scott W.R.; Roth G.; & Rivest J.F.; "View Planning for Automated 3D Object Reconstruction Inspection", *ACM Computing Surveys*, vol. 35, no. 1, pp. 64-96, 2003.

[21] Scott, W.R.; "Model-based view planning", *Machine Vision and Applications*, Springer-Verlag, 2007.

[22] Soucy, G.; Callari, F. & Ferrie, F.; "Uniform and complete surface coverage with a robot-mounted laser rangefinder". *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 1682–1688*. Victoria, B.C., Canada, 1998.

[23] Vasquez-Gomez, J. I.; Sucar, L.E. & Murrieta-Cid, R.; "View planning for 3d object reconstruction with a mobile manipulator robot". In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4227-4233, Chicago, Illinois, USA, Sept. 2014a.

[24] Wu, Z.; Song, S.; Khosla, A.; Tang, X.; Xiao, J.; "3D ShapeNets for 2.5D Object Recognition and Next-Best-View Prediction". Princeton University, MIT & CUHK, Sep 2014.

[25] Microsoft; ""Project Natal" 101". June 1, 2009. Archived from the original on June 1, 2009. [Online] Available at:

http://blog.seattlepi.com/digitaljoystick/2009/06/01/
e3-2009-microsoft-at-e3-several-metric-tons-of-press-releaseapalloza/

[26] Demerjian, C.; "A long look at Microsoft's XBox One Kinect sensor", Oct 15, 2013. [Online] Available at:

https://semiaccurate.com/2013/10/15/
long-look-microsofts-xbox-one-kinect-sensor/

[27] Lau, D.; 2013. Daniel Lau's Blog - The Science Behind Kinect. [Online] Available at:

http://lau.engineering.uky.edu/2013/11/27/the-science-behind-kinect/

[28] KUKA; Specifications manual. [Online] Available at:

https:
//www.kukakore.com/wp-content/uploads/2012/07/KUKA_LBR4plus_ENLISCH.pdf

[29] Sucan, I.A. & Chitta, S.; "MoveIt!", [Online] Available at:

http://moveit.ros.org

[30] Burgard, W.; Stachniss, C.; Bennewitz, M.; Arras, K.; "Robot Motion Planning", Slides from Mobile Robotics Course, Uni Freiburg, Germany. [Online] Available at:

http://ais.informatik.uni-freiburg.de/teaching/ss11/robotics/slides/
18-robot-motion-planning.pdf

[31] Teanby, N.A; Uniform grid algorithm, MATLAB implementation. [Online] Available at:

https://es.mathworks.com/matlabcentral/fileexchange/28842-grid-sphere

[32] Kirk, J.; Dijkstra algorithm, MATLAB implementation. [Online] Available at:

https://es.mathworks.com/matlabcentral/fileexchange/
20025-dijkstra-s-minimum-cost-path-algorithm

[33] Calli, B.C.; Singh, A.; Walsman, A.; Srinivasa, S.; Abbeel, P.; Dollar, A.M.; "The YCB Object and Model Set: Towards Common Benchmarks for Manipulation Research". [Online] Available at:

http://ycb-benchmarks.s3-website-us-east-1.amazonaws.com/

[34] CloudCompare: 3D point cloud and mesh processing software, Open Source Project. [Online] Available at:

http://www.danielgm.net/cc/

[35] Monica, R.; *Kinect Fusion* (ROS) GitHub repository. [Online] Available at:

https://github.com/RMonica/ros_kinfu

[36] Institute for Artificial Intelligence - University of Bremen. *IAI_Kinect_2: Tools for using the Kinect v2 in ROS*, GitHub repository. [Online] Available at:

https://github.com/code-iai/iai_kinect2

[37] Rebull, J.; Aalto GitLab of this thesis' reconstruction system. [Online] Available at:

https:
//version.aalto.fi/gitlab/rebullj1/aalto_reconstruction_pipeline.git

Appendices

A Planner GUI working guide

In this annex the procedure to perform an object scanning is explained in greater detail, as well as the different functionalities that the GUI offers to the user.

First of all the ROS source workspace has to be downloaded from the GitLab repository [37] and installed using the *catkin_make* command. This repository also provides the necessary MATLAB scripts and the compiled files to use custom ROS messages. The installation procedure is detailed in the repository's *README.md* file. On the other side, the KUKA LWR 4+ has to be correctly set with its tool calibration for the Kinect holder and the necessary scripts to run the *FRI* connection with the computer through the Ethernet cable.

The procedure to start the whole system is the following:

1. Place the object to scan in the scanning area, and put some other objects outside it to help the ICP tracking algorithm.

2. Turn on the KUKA robot and start the script to communicate with ROS through the *FRI* library.

3. Start the ROS system with the provided *roslaunch*. There are two main launches to execute inside the planner package:

   - *kuka_kinfu_sim.launch*: To execute the Gazebo simulation.
   - *kuka_kinfu.launch*: To run the real robot reconstruction system.

4. Start MATLAB suite and run the initialization script, called *sphere_server_init.m*. If the MATLAB ROS server was running before, it is necessary to shut it down first with MATLAB's *rosshutdown* command.

After executing this sequence different windows will appear with certain information and functionalities:

- *Rviz* (Figure 49): This program shows the status of the robot in a graphical way, so it is very helpful to understand the status of the reconstruction as well as the positions of the robot. It is composed of 3 spaces:

  1. The main graphical window, where the 3D model of the KUKA robot is represented and updated with its current position. This model also shows the possible viewpoints to locate the camera, which are numbered to allow an easy reference, and the space where the object will be scanned. Some other information can be shown in this 3D space, which is explained in the next pages.
  2. The 3D model current view of the reconstructed model. This window shows the *kinfu* output of the stored 3D model in real time.
  3. Other topic information, that can be selected to appear in the graphical window and can be useful for testing.

- *rqt_reconfigure*: This GUI allows the user to change important parameters at runtime. The system has been thought to be controlled with this windows, so they offer plenty of different functionalities to perform all kind of reconstruction experiments. Five classes with their own independent dynamic reconfigure windows have been defined:

  – *Auto_Init* (Figure 50): Contains all the higher level actions to perform the reconstruction tasks. This is the only window which the final user needs to modify to perform the general operations.

  – *Kinfu_Command* (Figure 51): Shows the different commands to change the *kinfu* node behaviour at runtime.

  – *Kinfu_Request* (Figure 52): Contains all the possible requests to send to the *kinfu* node.

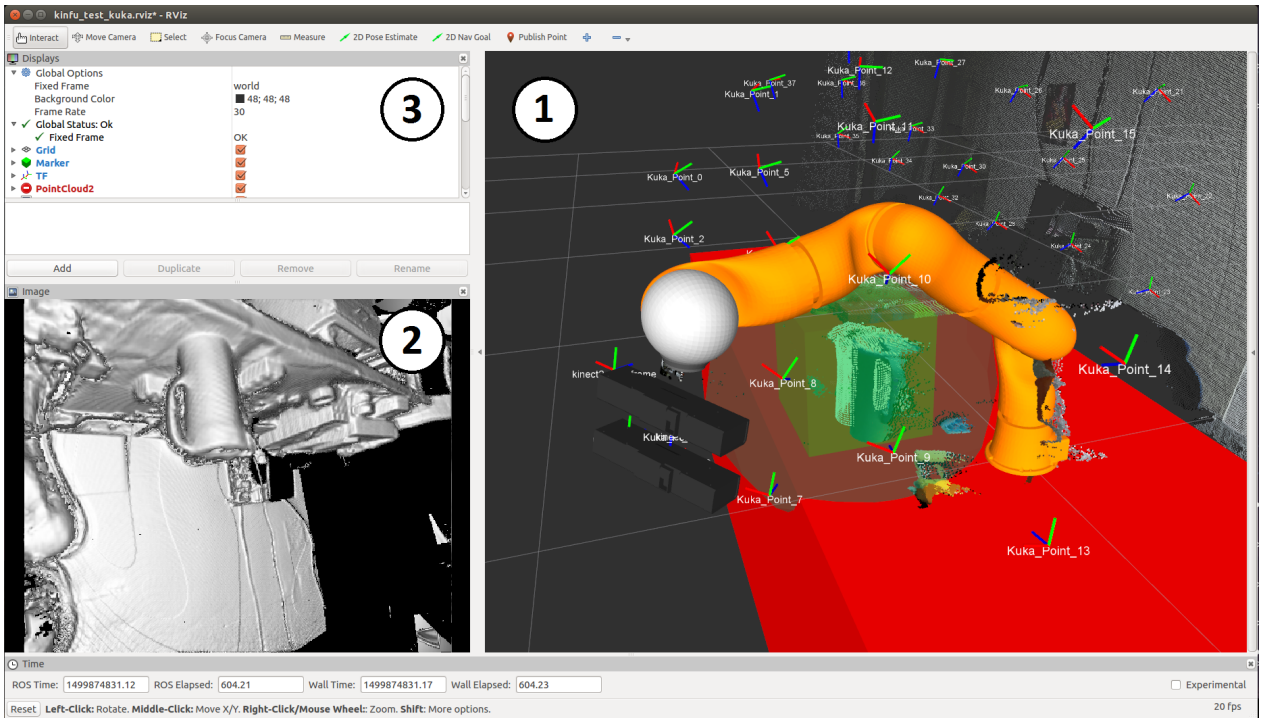


Figure 49: *Rviz* window of the program

  - *Kuka_Op* (Figure 53): Allows to send some actions to the KUKA robot for testing purposes.
  - *View_Points* (Figure 54): This window contains different options to configure viewpoints in the space and change the MATLAB service parameters.

- *Execution command line and MATLAB interface*: These windows are not necessary for usually performed utilities, but they show some information that can be useful for testing.

In the following sections all the different functionalities of the *rqt_reconfigure* windows are explained in more detail.

A.1 *Auto_Init window*

Auto_Init class automates the procedure to perform the reconstruction tasks. The different configuration options of this window are:

- *auto_settings (bool)*: If *True*, the predefined configurations of the *Auto_Init* class are taken into account. If *False*, this class does not have any effect and the tasks need to be performed manually with the other classes configurations.
- *require_MATLAB (bool)*: If *True*, the program will be blocked until the MATLAB services are available.
- *START (bool)*: If *True*, the reconstruction task will begin. If *False*, the KUKA robot will go to the initial pose defined in the *init_pose* parameter, and reset the current 3D model.
- *init_pose (int)*: Slider to select the initial pose number to start the reconstruction. The pose numbers are determined by the uniform sphere returned by the MATLAB service, that can be visualized in the *Rviz* interface.
- *discrete_reconstruction (bool)*: If *True*, the reconstruction is performed just in the viewing points, so the *kinfu* node is deactivated in the translations and activated again when the view point is reached. If *False*, the reconstruction is performed all the time.
- *forced_tf (bool)*: If *True*, the *kinfu* reconstruction reference is forced to stick to the Kinect frame given by the *MoveIt!* node, deactivating the ICP algorithm. If *False*, the reference uses ICP.
- *planner_type (int)*: This drop-down menu let you choose between different planning algorithms in order to test their performance (explained in Section 2.4.2 and 2.4.3):
  - *Next_Best_View (0)*: Goes directly to the next best view.

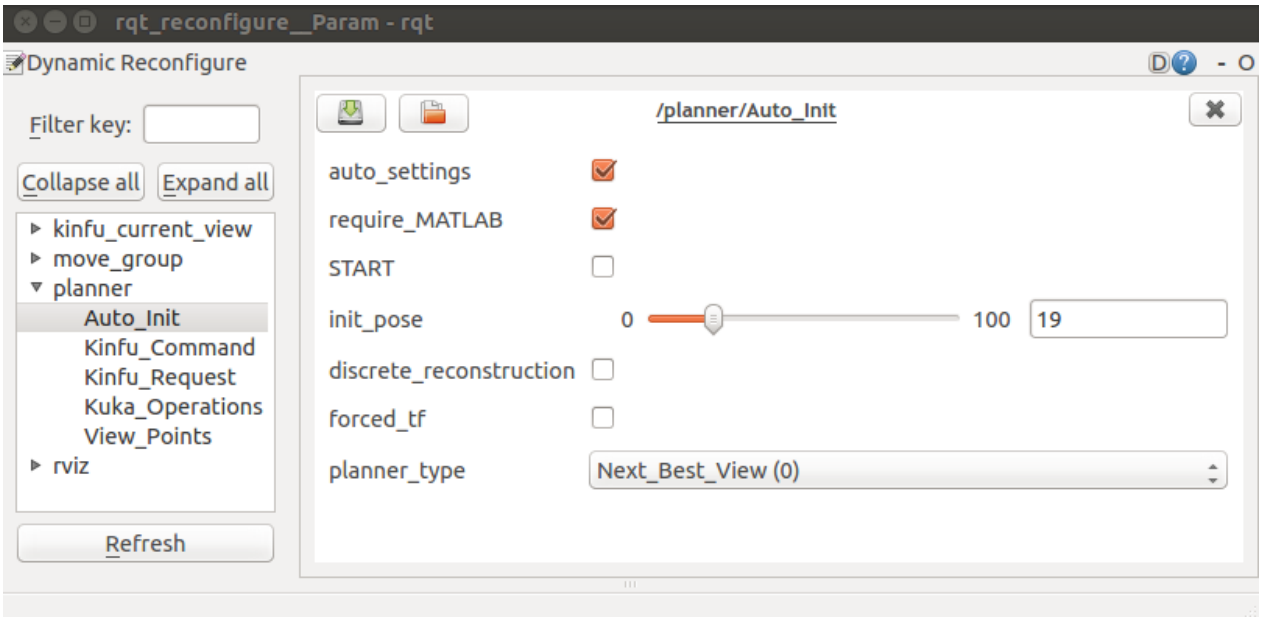


Figure 50: *rqt_reconfigure* window to modify the *Auto_Init* class parameters

– *Dijkstra_Dynamic (1)*: Asks the Dijkstra path to reach the next best view. This algorithm goes to the next viewpoint of the received Dijkstra path, and when reaches it, computes the Dijkstra path again pointing to the updated next best view.

– *Dijkstra_NBV (2)*: Asks the Dijkstra path to reach the next best view, and does not compute Dijkstra again until it finishes this path.

– *Dijkstra_NBV_Dynamic (3)*: Asks the Dijkstra path to reach the next best view. This algorithm goes to the next viewpoint of the received Dijkstra path, and when reaches it, computes the Dijkstra path again pointing to the same old objective.

A.2 *Kinfu_Command window*

Kinfu_Command sends different commands to the *kinfu* reconstruction node at runtime. The available commands are:

- *kinfu_active*: If *True*, the reconstruction system is started; if *False* it pauses the node.
- *kinfu_reset*: If *True*, it sends a command to reset the node.
- *set_enabled_min_movement*: Enables or disables the minimum movement required to update the *kinfu* representation (not used in this project).
- *forced_tf*: Enables or disables the external tracking using the defined TF in the *kinfu* node. If the *forced_tf* is activated, *ICP* tracking is automatically disabled and vice-versa.
- *clear_sphere*: If *True*, the reconstructed model inside the sphere defined with the following sliders is cleared.
- *clear_bounding_box*: If *True*, the reconstructed model inside the bounding box defined with the following sliders is cleared. These sliders contains two points that defines the bounding box extremes.
- *kinfu_trigger*: If the *kinfu* is suspended, this will execute it for a single frame.

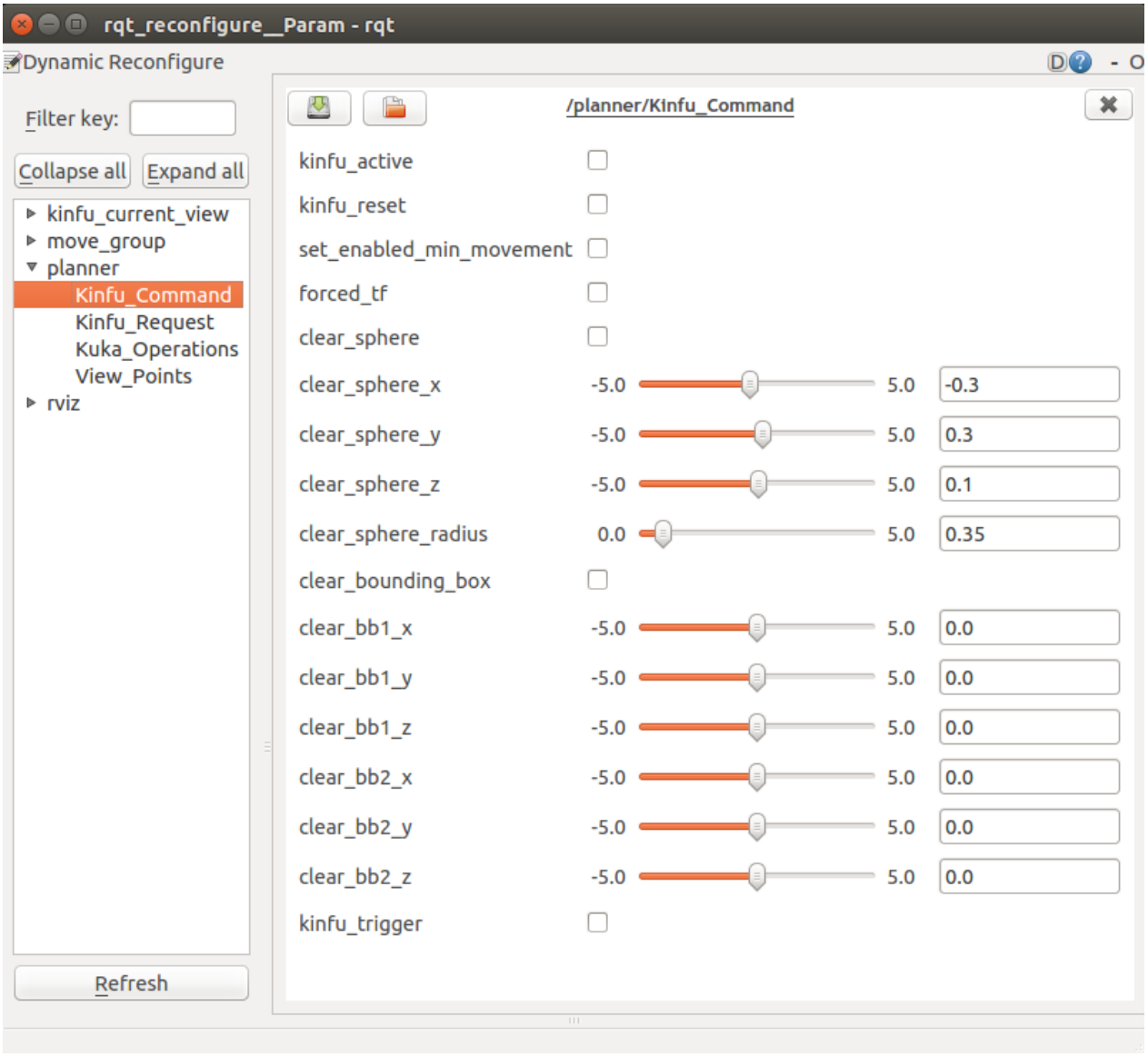


Figure 51: *rqt_reconfigure* window to modify the *Kinfu_Command* class parameters

A.3 *Kinfu_Request window*

Kinfu_Request sends different commands to the *kinfu* reconstruction node at runtime. The available requests are listed in the request drop-down menu, and they are periodically sent when the *send_request = True*. All the other windows options will be added to the sent request, and might have no effects if they are not required for a certain action. The request list is the following:

0. *PING*: Sends a request to test if the call and response system is working. It is useful to set *request_reset = True* option with this request, so the *kinfu* system is reset without outputting unnecessary information.

1. *GET_TSDF*: Asks the *kinfu* node to return the current TSDF volume of the reconstructed system. This feature is not used in our application.

2. *GET_MESH*: Asks the *kinfu* node to return the current mesh (a *pcl_msgs/PolygonMesh* message) of the reconstructed object. After getting the mesh, *Kinfu_Request* class stores it in a *.vtk* file in the *.ros* folder.

3. *GET_CLOUD*: Asks the *kinfu* node to return the current pointcloud (*sensor_msgs/PointCloud2* message) of the reconstructed system. As the output of this application is a mesh (it holds more information), this pointcloud is only used for visualization purposes in *Rviz*.

4. *GET_KNOWN*: Asks the *kinfu* node to return the known voxels of the reconstructed system. To use this the internal *kinfu* parameter *extract_known_points* has to be set to *True*, but this feature is not used in our application.

5. *GET_VIEW*: Simulates a Kinect view from a specific pose and outputs it as an image message. This feature is not used in our application.

6. *GET_VIEW_CLOUD*: Simulates a Kinect view from a specific pose and returns a pointcloud with knowledge information (0: unknown, 1 occupied). This pointcloud is shown in *Rviz*, but this feature is not used in our application.

7. *GET_VOXEL_COUNT*: Asks the *kinfu* node to return the known and unknown voxels of the reconstructed system from several simulated views. This request gives as output an array (*std_msgs/UInt64MultiArray*). If more than one viewpoint is specified, it returns the number of known / unknown voxels for all the views. This request is the base of the planning algorithms used in this project because it gives information about each viewpoint quality. If no views are specified, only one global pair is returned.

8. *GET_VOXEL_GRID*: Returns a 3D array of float values inside the bounding box, which is mandatory. This feature is not used in our application.

9. *GET_FRONTIER_POINTS*: Returns the frontier, defined as the contact surface between empty and unknown voxels, as points with normals. The *kinfu* internal parameter *extract_incomplete_frontier_points* has to be set to *True*. This feature is not used in our application.

10. *GET_BORDIER_POINTS*: Returns the border of the mesh as points with normals. The *kinfu* internal parameter *extract_incomplete_border_points* has to be set to *True*. This feature is not used in our application.

The other windows options are the following:

- *request_transformation*: If *True*, an affine transformation defined in the code will be applied to the points during extraction. This is NOT ALLOWED for *GET_TSDF* request type. This feature is not used in our application.

- *request_bounding_box*: If *True*, the mesh or pointcloud is cropped before publishing. The next sliders set a minimum and maximum value for each point coordinate which defines the bounding box. Only points or triangles inside the bounding box are published. The

transformation from *request_transformation* is applied before cropping. Note that this option is only useful to reduce ROS message size. In the current implementation, *kinfu* always processes the whole pointcloud even if only a part is needed.

- *request_bounding_box_view*: For view operations, treat all the points outside the bounding box defined by the sliders as empty. It is useful to exclude noisy areas at the edges of the observed world, but in our application is not used.

- *request_remove_duplicates*: Removes duplicate points if the request type is *GET_CLOUD* or *GET_MESH*.

- *request_sphere*: If *True*, only points inside the slider defined sphere will be used. For now, this feature is implemented only for the *GET_VOXEL_COUNT* request, so it is always used to define the sphere where the object is located.

- *request_subsample*: If *True*, the resulting pointcloud will be subsampled according to the next slider. However, as it is currently implemented only for *GET_KNOWN* requests, this feature is not used in our application.

- *tsdf_center_distance*: Used in *GET_VIEW* request, it is the distance from the sensor to the centre of the *TSDF* volume. This parameter also affects the shifting distance of the *TSDF* volumes during the reconstruction.

- *request_camera_intrinsics*: It sets the camera intrinsics for calibration, defined inside the code. It is used for *GET_VIEW*, *GET_VIEW_CLOUD* and *GET_VOXEL_COUNT* requests types.

- *request_view_poses*: If it is *True*, an array of poses is attached to the request so the *GET_VOXEL_COUNT* request simulates the views with them in order to get the results. This array of poses is generated in the *View_Points* class, which can generate from single testing poses to a set of viewpoints using the MATLAB services.

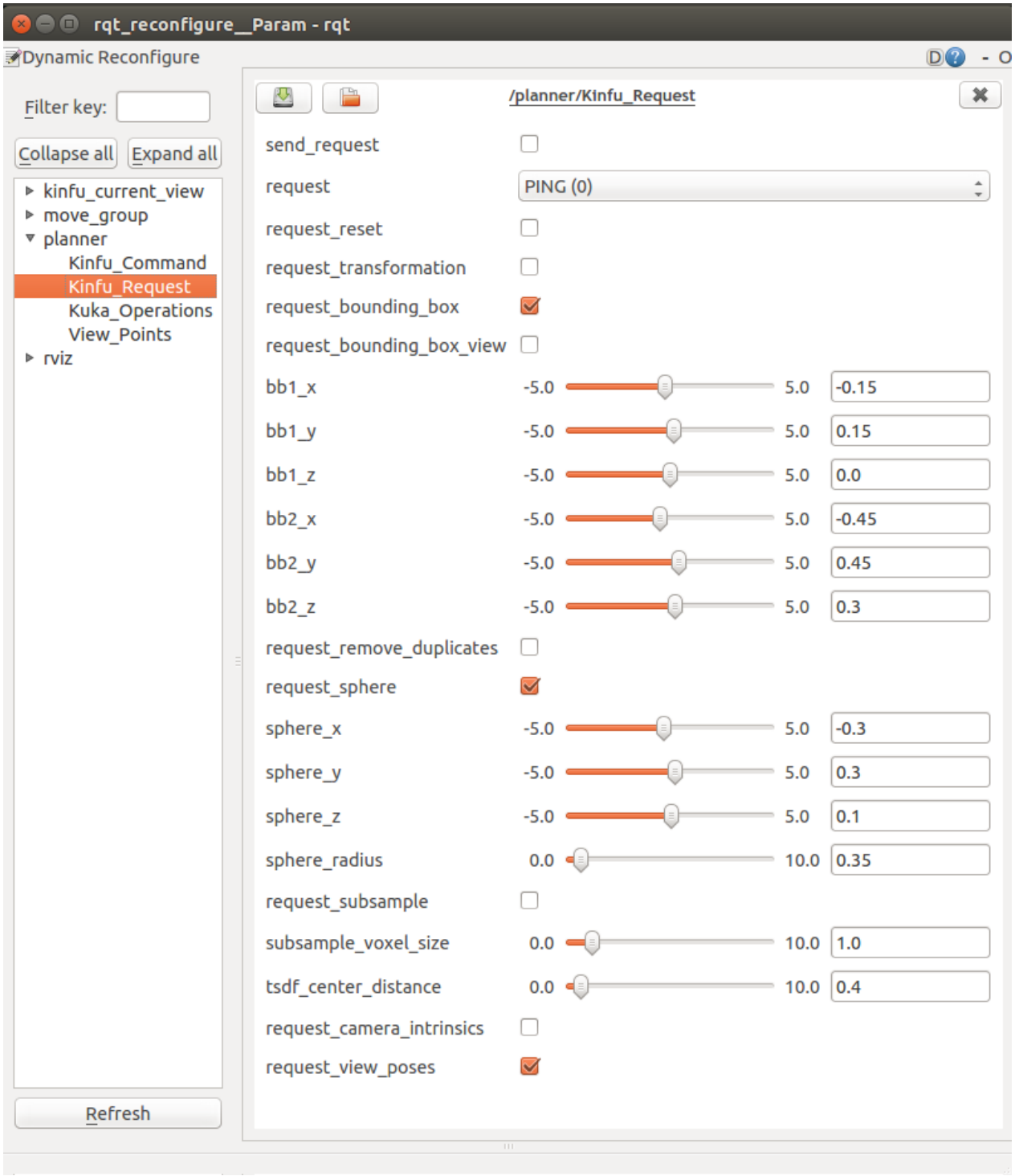


Figure 52: *rqt_reconfigure* window to modify the *Kinfu_Request* class parameters

A.4 *Kuka_Operations window*

This window is the responsible to configure the robot arm movements and actions, as well as the view planner algorithms management. The different options are:

- *simulated_voxels*: If *True*, the program internally simulates known and unknown voxels numbers for each viewpoint. The best view is placed in the *simulated_nbv* slider pose number. By this way, this option is useful to test the view planner algorithms without the need of *kinfu* or the real robot, as this can be used also with the *Gazebo* simulation.

- *kuka_pose_num* and *send_kuka*: If *send_kuka = True*, the robot will move to the *kuka_pose_num* viewpoint.

- *NBV* and *dijkstra*: Manually activates the NBV or *dijkstra* planning algorithms. Note that the following instructions need to be followed to make the system work:

  - In *Kinfu_Command*: Set *kinfu_active = True*
  - In *Kinfu_Request*: Set the *request* parameter to *GET_VOXEL_COUNT (7)* and set *send_request = True* to send the request periodically.
  - Also in *Kinfu_Request*: Check that the *request_sphere* is well defined and *request_view_poses = True*.
  - In *View_Points*: Check that the correct viewpoints are being printed in the *Rviz* window, and if not, configure them before launching the planning algorithms.

- *tilt_amplitude*: The KUKA arm has been programmed to tilt the camera every time it receives the same viewpoint as where it is located. This feature allows the *kinfu* node to better reconstruct the model if the view has complex shapes. This parameter sets the amplitude, in degrees, of this tilting movement.

- *vox_threshold*: When performing the reconstruction, this is the threshold that defines the maximum allowed unknown voxels of the 3D model to consider the scanning

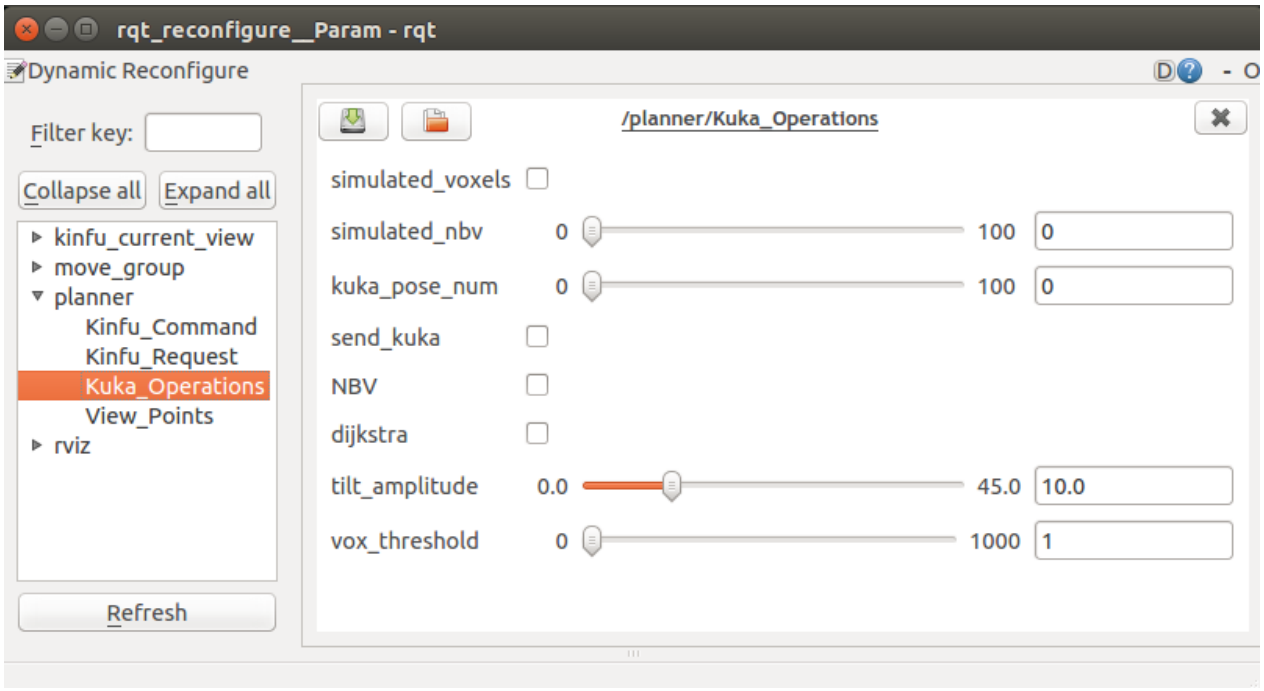


Figure 53: *rqt_reconfigure* window to modify the *Kuka_Op* class parameters

complete. Thus, this parameter is related with the final model quality but also with the scanning time.

A.5 *View_Points window*

This window is in charge of the poses and viewpoints generation. The generated viewpoints are plotted in *Rviz* with their reference number, which is needed in some other functionalities such as in *Kuka_Operations*. The different parameters of this window are:

- *view_sphere*: Different parameters of this sphere can be modified through the sliders. The view sphere is the virtual sphere where all the viewpoints are located pointing always to the centre. If *same_sphere = True*, this view sphere will be the same as the defined in *Kinfu_Request*.

- *sphere_rotation_z*: This parameter sets the final orientation of the viewpoints, which is relevant to allow the correct positioning of the camera far from the robot joint limits.

- *sphere_offset*: Difference between the object sphere and the viewing sphere: $radius = view\_sphere\_radius + sphere\_offset$.

- *n_sampl*: Number of uniform sampled points in the viewing sphere. This parameter refers to viewpoint density, and it is sent to the MATLAB function through the *sphere_sampling* service.

- *ws_radius*: Radius that defines the working space of the KUKA robot. Viewpoints outside this sphere are not taken into account.

- *limit_radius*: Radius that defines the sphere which excludes the viewpoints below the table surface or too close to it.

- *see_all_poses* / *see_iterating_poses*: Visualization options in *Rviz*. The first one shows all the viewpoints at once whereas the second only shows one different at a time, iterating through all of them.

- *poses_type*: This drop-down menu allows to choose between different viewpoint options, where some of them will need the next slider parameters to be manually set.

0. *Manual_2D*: This option sets a single viewpoint with the position (x, y, z) and orientation ($\varphi = Roll$, $\theta = Pitch$, $\varphi = Yaw$) defined in the following sliders.
1. *Manual_Sphere*: This option sets a single viewpoint pointing towards the viewing sphere centre. The orientation sliders (φ, θ, φ) locates the viewpoint so the positioning sliders does not work. This option uses the *pose_in_sphere* MATLAB service to set the position and orientations from this parameters. This feature is very useful for testing purposes as it makes it very easy to create an arbitrary reachable viewpoint for the KUKA robot.
2. *Uniform_Sphere*: This is the used option for the final application, as it calls the MATLAB *sphere_sampling* service and returns all the viewpoints placed in the viewing sphere.

Note: All the angles are defined in degrees.

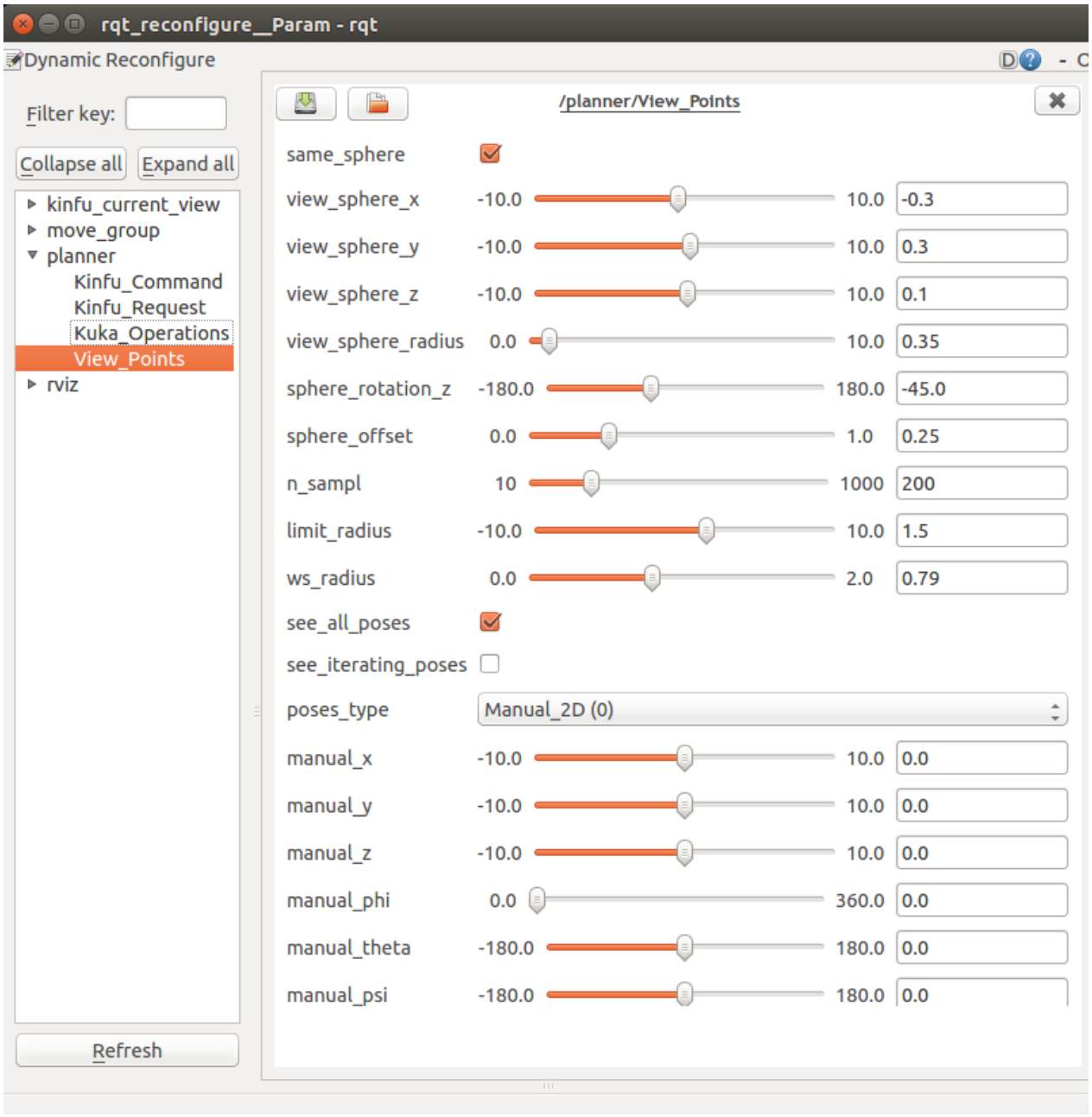


Figure 54: *rqt_reconfigure* window to modify the *View_Points* class parameters

B TF scheme tree

In this annex the TF structure of the system is shown through the ROS *view_frames* utility. This is useful to understand how the different frames are related in this application. As the global TF scheme tree is very big and can not fit in a single page, it is divided in 3 different sections depending on their nature.

B.1 Reference frame

Figure 55 shows the beginning of the TF tree, that grows from the *world* frame. On the right side there are all the frames related to the *Kinect Fusion* algorithm. As the *kinfu* node process the different viewpoints, they are referenced to the *kinfu_first_frame* TF. In this plot only a single viewpoint is shown for simplicity (*Kuka_Point_0*), but in the final application all the viewpoints would appear in this area. On the left side the *MoveIt!* links of the environment model are shown (*box* and *sphere*) as well as the beginning of the robot model tree starting from its base frame (*lwr_base_link*).

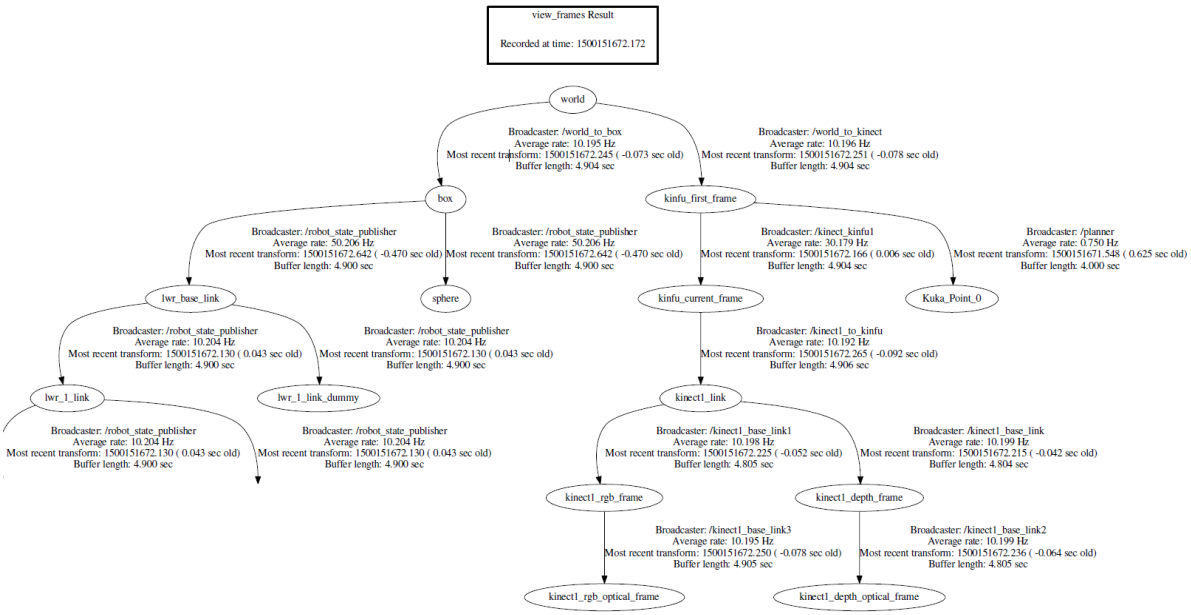


Figure 55: TF tree of the reference frames and *kinfu*

B.2 KUKA frame

Figure 56 shows the TF tree of the KUKA model defined in *MoveIt!*. Every frame is the link between two joints of the KUKA robot. This tree grows from the *lwr_base_link* shown in the Figure 55, until the *lwr_7_link* that corresponds to the tool centre point where the Kinect Holder is being attached.

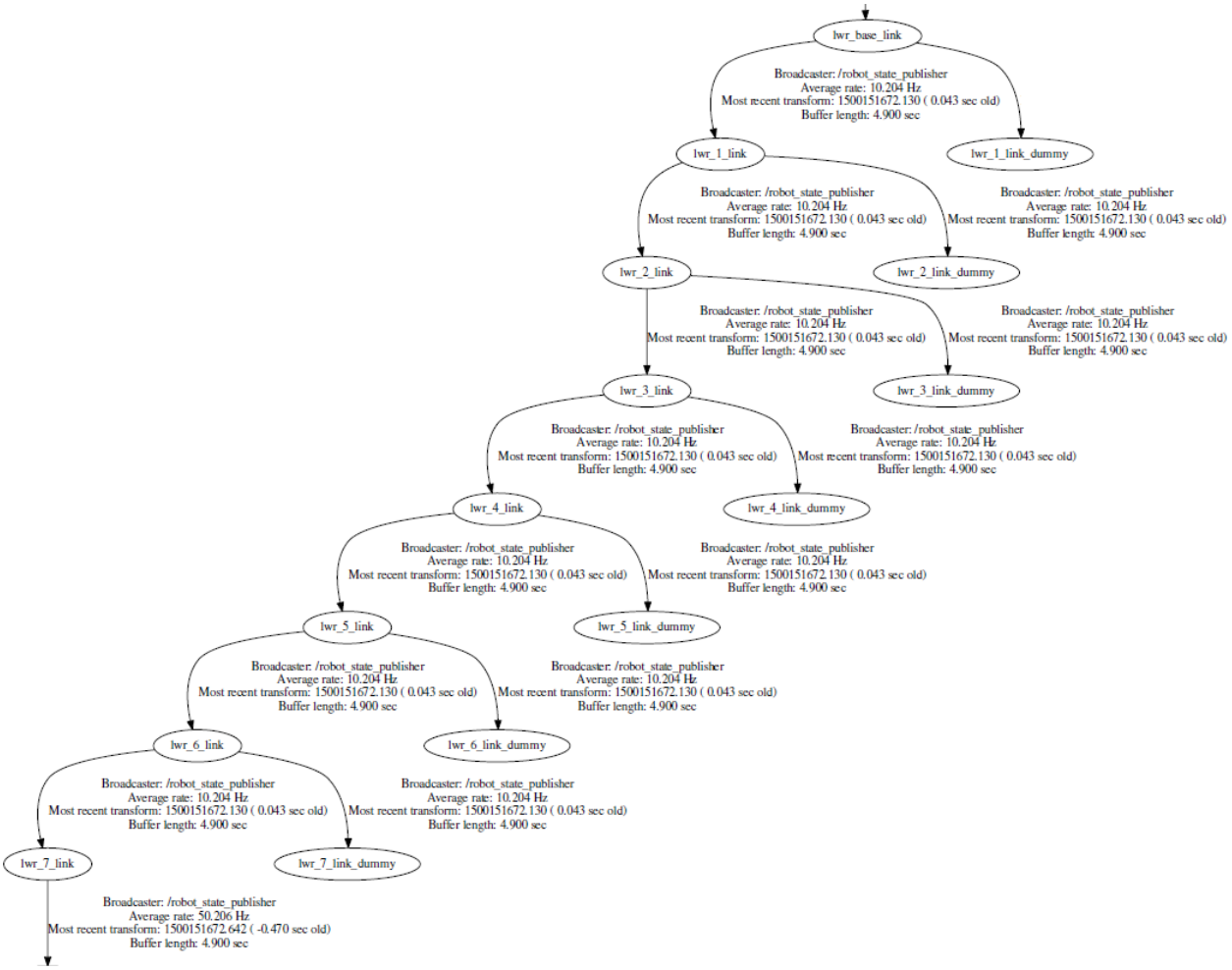


Figure 56: TF tree of the *MoveIt!* KUKA model frames

B.3 Kinect Holder frame

Figure 57 shows the TF tree of the Kinect holder model defined in *MoveIt!*. It grows from *lwr_7_link*, that corresponds to the tool centre point of the KUKA robot until the two different Kinect reference frames: *kinect1_link_frame* and *kinect2_link_frame*. This two frames are used to build the model using the *forced_tf* option. Its parent frame *kinect_ee*, that is located between the two Kinect cameras, is used as the end effector to place the cameras in the required viewpoints.

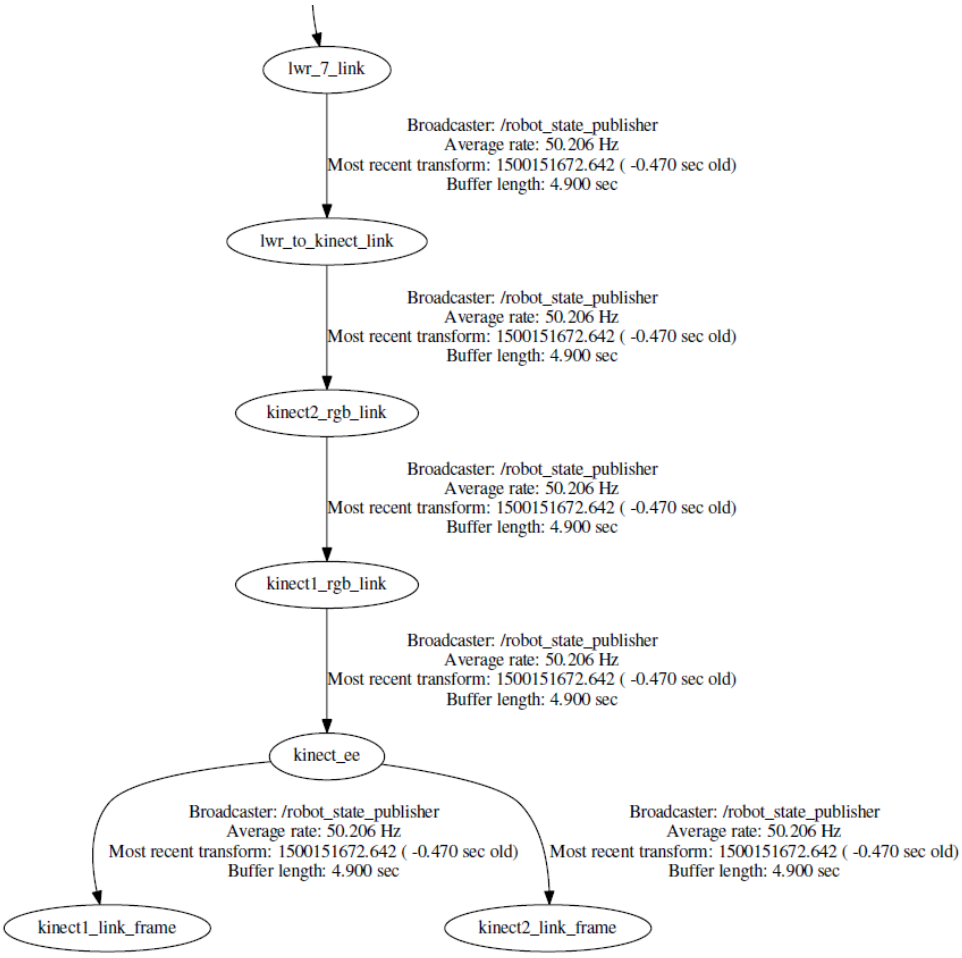


Figure 57: TF tree of the *MoveIt!* Kinect holder model frames